WMP

Wireless MAC Processor
Enjoy programming your MAC

# DOCUMENTATION

http://wmp.tti.unipa.it

Work carried out under the EU Project FLAVIA - Grant 257263

# Contents

**1 The Wireless MAC Processor: General concepts 1**

1.1 Introduction . . . 1

1.2 Organization of this Document . . . 1

**2 The Wireless MAC Processor at a glance 3**

2.1 Introduction . . . 3

2.2 MAC abstraction layers . . . 3

2.3 Elements of a FSM . . . 3

2.3.1 Binary-Byte-Code . . . 4

2.4 WMP-Editor . . . 5

**3 WMP details: events, conditions and actions 7**

3.1 Events . . . 7

3.2 Conditions . . . 8

3.3 Actions . . . 9

3.4 State parameters . . . 11

**4 WMP-Editor 13**

4.1 Overview . . . 13

4.2 Features . . . 13

4.3 Editor Description . . . 13

4.3.1 WMP Machines . . . 14

**5 MAC Design 19**

5.1 Introduction . . . 19

5.2 Distributed Coordination Functions (DCF) . . . 19

5.2.1 Reception operation mode . . . 19

5.2.1.1 Don't ack frame . . . 21

5.2.1.2 Do ack frame . . . 21

5.2.2 Transmit operation mode . . . 21

5.2.3 State Parameters in DCF . . . 22

5.2.4 DCF programmability . . . 22

5.3 Time Division Multiple Access (TDMA) . . . 23

5.3.1 State Parameter in TDMA . . . 23

5.4 Direct Link . . . 23

5.4.1 DLS . . . 23

5.4.2 DCLS . . . 23

5.4.3 State parameter in D(C)LS . . . 25

**6 Byte-Code details 29**

6.1 Introduction . . . 29

6.2 Structure of the Byte-Code . . . 29

6.3 Byte-Code of the DCF State Machine . . . 30

6.4 Binary-Byte-Code: details . . . 32

6.4.1 States region . . . 33

6.4.2 Transitions region . . . 33

6.4.3 State parameters region . . . 34

**7 Byte-Code-Manager 36**
7.1 bytecode-manager options list . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
7.2 Stand-alone Operation Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
7.2.1 Byte-Code injection and activation . . . . . . . . . . . . . . . . . . . . . . . 36
7.2.2 Delayed Byte-Code switching . . . . . . . . . . . . . . . . . . . . . . . . . . 37
7.2.3 Client-Server Operation Mode . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**8 Usage scenarios and examples 40**
8.1 Basic usage at a glance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
8.2 Advanced usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
8.3 Quick usage examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
8.3.1 Example 1: Byte-Code load and activation . . . . . . . . . . . . . . . . . . . 41
8.3.2 Example 2: delayed Byte-Code activation . . . . . . . . . . . . . . . . . . . . 42
8.3.3 Example 3: accurate delayed Byte-Code activation . . . . . . . . . . . . . . 42
8.3.4 Example 4: remote Byte-Code switching . . . . . . . . . . . . . . . . . . . . . 43
8.3.5 Example 5: remote Byte-Code delayed switching . . . . . . . . . . . . . . . 44

**A APPENDIX 45**
A.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
A.2 Modify the kernel to improve usability and stability of the testbed environment . . 45
A.2.1 Avoid deassociation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
A.2.1.1 For older kernel versions . . . . . . . . . . . . . . . . . . . . . . . . 45
A.2.1.2 For more recent kernel versions . . . . . . . . . . . . . . . . . . . . 45
A.2.2 Solving the throughput problem of Iperf . . . . . . . . . . . . . . . . . . . . 46
A.3 How to enable “debugfs” and “b43 debugging” in the Kernel . . . . . . . . . . . . . 46

# Chapter 1

# The Wireless MAC Processor: General concepts

## 1.1 Introduction

The **Wireless MAC Processor (WMP)** is an architecture platform devised to run a wireless MAC program defined in terms of a **Finite State Machine (FSM)**. It has been shown, in fact, that MAC protocols can be described in terms of state machines made of three main elements: actions, events and conditions. In the WMP case, actions are commands for the radio hardware, such as transmit a frame, set a timer, and switch to a different frequency channel. Events include hardware interrupts such as channel up/down signals, indication of reception of specific frame types, expiration of timers and so on. Conditions are boolean expressions evaluated on internal configuration registers that can either explicitly updated by actions, or implicitly updated by events. Some registers are store general MAC layer information (like the current radio channel or the power level), or more specific MAC variables (like the contention window value and the backoff parameter). Starting from an initial (default) state, the WMP waits for events which trigger state transitions. The actual transition can be enabled or disabled by verifying a boolean condition, while an action on the hardware system (i.e. on the transreceiver) can be performed before completing the transition to the new state.

For these reasons, the WMP differs from off-the-shelf wireless NICs powered with their "vanilla" code: while the latter are tied to a specific MAC protocol (i.e., IEEE 802.11), the WMP architecture can run generic FSM, hence it can implement users' designed MAC programs. On the basis of a pre-defined (hardware-dependent) set of actions, events and conditions which represent the platform API, a MAC programmer can easily compose different channel operations into a MAC program and execute it on the WMP.

## 1.2 Organization of this Document

This document describes the WMP that was developed by the CNIT research team (under the EU project FLAVIA) following the paradigm introduced above, on a specific commercial wireless card designed by Broadcom. The WMP has been implemented by writing a new firmware that replaces the original sofware from Broadcom with a generic state machine executor called **MAC-Engine**: this work has been possible thanks to the availability of a documented open firmware for a specific chipset of the big AirForce54G family of Broadcom wireless NICs, namely the OpenFWWF Project [1].

The document also describes the API available on this platform (i.e. the list of events, actions and conditions to be used for defining MAC programs) and some tools for developing and debugging MAC state machines, including:

- **WMP-Editor**, a graphical tool, working as an editor for describing a MAC program in terms of a graphical representation of state transitions and state labels;
- **WMP-Compiler**, a compiler integrated into the WMP-Editor that can translate the graphical representation of the MAC program into a textual transition table and into a coded representation that can be actually loaded into the NIC (the Byte-Code);

[1]OpenFirmWare for WiFi networks, http://www.ing.unibs.it/openfwwf

- **Byte-Code-Manager**, a tool for reading a Byte-Code and injecting it into the WMP.

The combination of the MAC-Engine, the WMP-Editor, the WMP-Compiler, the Byte-Code-Manager and the driver is a complete and cheap tool-chain that allows developing and testing new MAC programs in a very simple, robust and quick way over an ultra-cheap platform. Each component of the toolchain can be found on the site http://wmp.tti.unipa.it, where we provide:

- this documentation;
- the MAC-Engine firmware that replaces the original card firmware;
- the graphical editor WMP-Editor;
- some Byte-Code examples (including standard DCF, Time Division Multiple Access and Direct Link);
- the Byte-Code-Manager.

Note that the current MAC-Engine firmware has been tested on BCM4311 and BCM4318 chipset revisions, using the B43 driver on Linux kernel 3.1.4 (for more information check the Appendix). The firmware supports both the infrastructure (working as a station) and the ad-hoc mode, it is compatible (in terms of protocol timings, frame fields, etc.) with legacy DCF stations in 11b and 11g mode, and it provides throughput performance comparable with the proprietary card firmware when executing the DCF state machine. It does not currently support: the RTS/CTS handshake (to be disabled when loading the b43 module), the hardware cryptography acceleration (to be used without encryption!) and the dot11 QoS mode (to be disabled when loading the module). Moreover, it has not been tested for working in 11a mode.

Some useful links for integrating this documentation can be found in the following list:

- WMP team: http://wmp.tti.unipa.it/
- OpenFWWF team: http://www.ing.unibs.it/openfwwf/
- BCM Specs Site: http://bcm-v4.sipsolutions.net
- B43 information: http://wireless.kernel.org/en/users/Drivers/b43#firmwareinstallation
- B43 compilation tools: http://git.bu3sch.de/git/b43-tools.git

For any doubt, please do not hesitate to contact the WMP team!

Chapter 2

# The Wireless MAC Processor at a glance

## 2.1 Introduction

The **Wireless MAC Processor (WMP)** is a Finite State Machine (FSM) executor that runs inside a wireless network interface card (NIC): having direct access to the underlying hardware functions of the NIC, Radio, PHY and other facilities like timers, the finite state machine running in the WMP can be tailored to mimic a full featured Medium Access Control algorithm. This is achieved by exposing to the WMP a number of basic elements directly connected to the hardware, such as signaling from the Radio and the PHY that reports incoming frames from the air, and a few elementary actions, like frame passing to/from the radio. This approach shows many improvements with respect to classic implementations:

- Users can easily define new MACs by composing finite state machines using a graphical tool: thanks to a drag and drop interface, simple operations like frame sending or ack waiting can be composed into a complex MAC;
- Existing MAC can be easily modified/updated;
- Different MACs can be used in the same machine, either they can be selected for independent execution or they can coexist in the WMP at the same time and periodically activated (virtualization).

The theoretical approach followed in the realization of the WMP is that of a FSM with essentials like states and transitions enriched with three additional elements, namely events, conditions and actions that improve the flexibility of the resulting system. We report in chapter 3 all the elements that can be used define several different MAC programs.

## 2.2 MAC abstraction layers

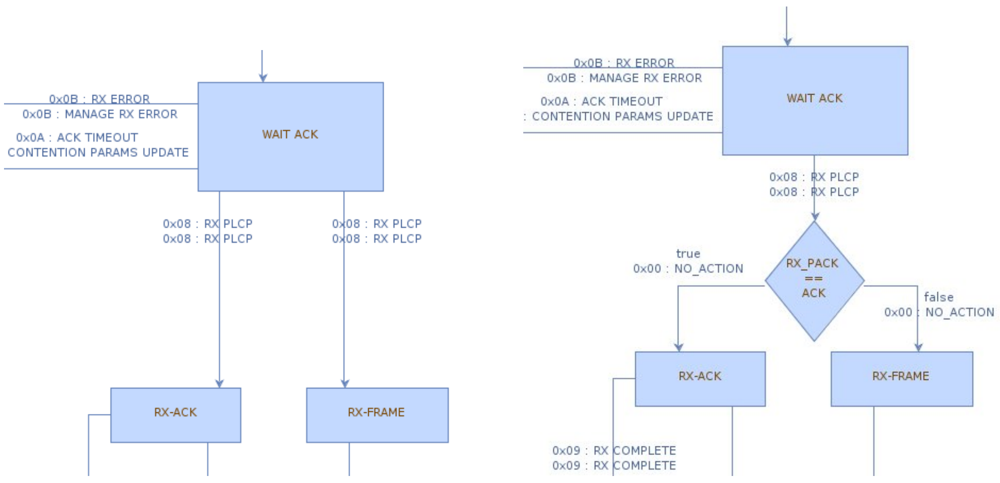
MACs defined for the WMP can be considered following two abstraction layers: a textual one, where everything is described using text expressions, and a graphical one, where the state machine is described through a practical graph based approach. The former representation is the **Byte-Code**, a text file that can be either written at hand by users, or automatically generated by the **WMP-Editor**, a graphical tool that can be used to build the latter representaion. This tool helps users composing new FSM, elements can be, in fact, connected together thanks to a straightforward drag and grop interface: furthermore it also checks for semantic errors during the design phase and for this reason it is strongly recommended to use it and avoid to manually write a Byte-Code. Independently of how the Byte-Code is obtained, users must run a compiler to convert it into a binary format, the **Binary-Byte-Code**, so that the resulting file can be executed by the WMP. The software for pushing the code to the MAC is called **Byte-Code-Manager**.

## 2.3 Elements of a FSM

Every FSM is characterized by the following elements: i) states ii) transitions between couple of connected states iii) events, conditions and actions associated to each transition. While events and conditions drive state transitions, actions are performed by the MacEngine during a transition and eventually implement MAC specific functions. All these "elements" are specified by users into the Byte-Code.

**Events** are like interrupts and are internally represented by some flags flipping when the corresponding hardware section detect a change in the PHY: e.g., TX READY reports to the MacEngine that a transmission that was scheduled in the past has been started. When in a given status, the WMP waits for the corresponding events, as specified in the Byte-Code, to occur. When one does, the transition can immediately take place, or one additional **Condition** can be evaluated, again as described in the Byte-Code. In the first case the transition is unique; in the second case, instead, there are two different transitions, triggered by the same event but having two different arrival states, depending on the value taken by the condition. Finally, **Actions** are executed during a transition to a new state and they generally represent elementary activities like frame receving.

For the sake of clarity we report in Figure 2.1 a state transition characterized by an event and a condition: on the left the we have two transitions from state 0x04, they are both triggered by the same event RX_PLCP but the one that is actually followed by the WMP depends on the value of condition [RX_PACKET == ACK]. The resulting graph involves only three states. The graph on the right, instead, involves four states even if the behavior is equivalent: here a conditional status is automatically introduced by the WMP-Editor to evaluate the condition.



(a) Implicit (b) Explicit

Figure 2.1: Transition with condition checking explained.

WMP checks events in polling: if one or multiple events are to be checked in a given state, they are all polled and the first that verifies triggers the corresponding transition which is unique. If no events verify than loop restarts. A condition, instead, is evaluated istantaneously and either one or the other of the associated transitions are followed by the WMP according to the value taken be the condition itself.

### 2.3.1 Binary-Byte-Code

Two independent Binary-Byte-Code s can be stored inside the WMP, each limited to 1Kbit: since switching time is negligible, this design choice enables immediate reconfiguration of the MAC algorithm so that two different behavior can be selected either automatically given user request or periodically. In the latter case switching time can be synchronized between stations. A user space tool called **Byte-Code-Manager** can be used either to inject Binary-Byte-Code to the WMP, to set parameters for tailoring the MAC behavior, or to monitor the WMP, so it can be used for

- Loading one Byte-Code in one of the two available slots
- Activating one Byte-Code
- Setting up timers for activation/shutdown of Byte-Code s
- Showing conditions for activation/shutdown by means of timers

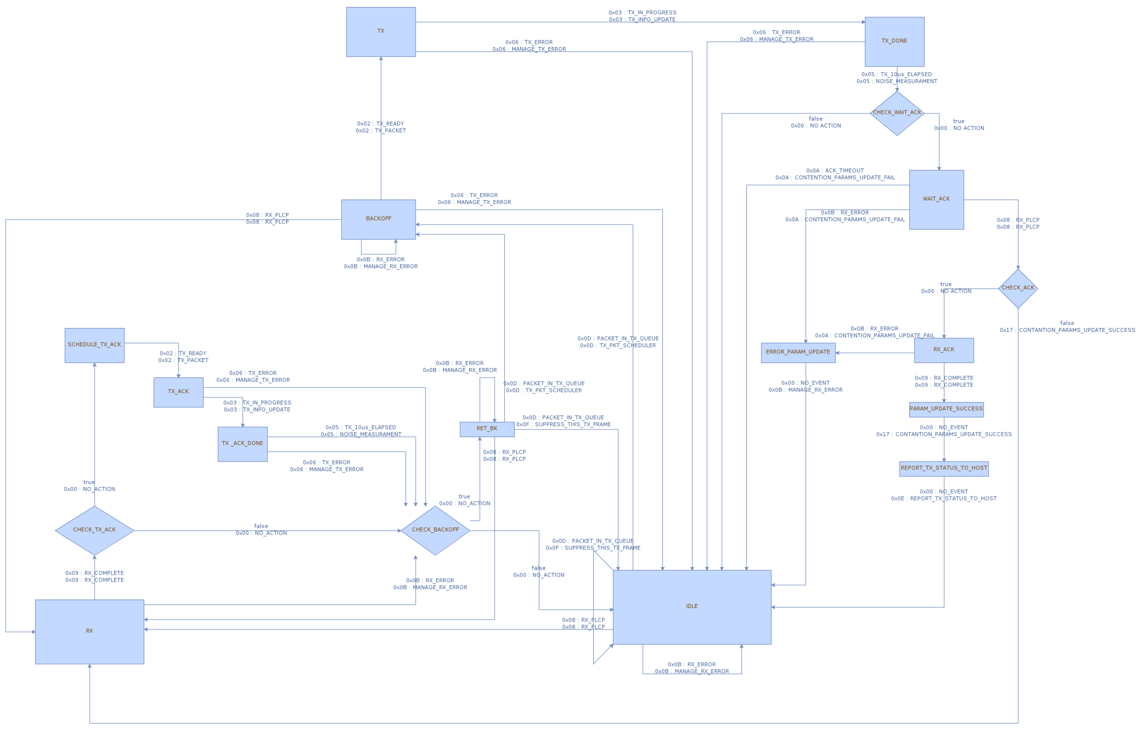


Figure 2.2: FSM implementing DCF, graphical representation.

- Managing a local WMP
- Managing a remote WMP

## 2.4 WMP-Editor

The WMP-Editor tool enables a easy and straightforward implementation of new MACs starting from their **Graphical Representation**. First, users place states in the project window and connect them with transitions. Then they can tailor each transition adding events, conditions and actions through pull down menus. Finally, when the MAC is ready, WMP-Editor can export it to the **Textual Representation**, which though being more compact is much harder to understand by users that do not know all specifics of the text format. Although one can start by writing the text file, it is always better to use the WMP-Editor: during the design phase, in fact, the tool does not allow semantic errors by construction while manual encoding might do. Before injection to the WMP memory, the project must be converted to the **Binary Representation**, namely the Binary-Byte-Code: states, transitions, events, conditions and actions are optimized in a logic that the WMP can understand.

Figure 2.2 reports a state machine that implement the Distributed Coordination Function (DCF) MAC algorithm: although this is an abstract representation, the WMP-Editor translates it into a running Binary-Byte-Code which actually implements DCF. More details about the graphical tool will follow in Chapter 4. Figure 2.3 reports an excerpt of the Byte-Code of the same state machine, Figure 2.4 the Binary-Byte-Code.

```
#state 0
000010
00F0
000006
010001000100$

#state 1
000010
03F4
000006
0E0101080508260101OB010B3A01010D0200$

#state 2
000010
0CF2
000006
5B01010E030D00000100010F$
```

Figure 2.3: Excerpt of the FSM implementing DCF, Byte-Code representation.

```
0000000: 00 00 ff ff  00 06 ff ff  ff ff ff ff  ff ff ff ff
0000010: ff ff 00 00  00 00 00 00  00 00 00 00  00 00 00 00
0000020: 00 00 00 00  00 00 00 00  00 00 00 01  00 00 00 01
0000030: 00 01 00 01  00 00 ff ff  00 1f 03 ff  00 1f 00 00
0000040: 00 00 0b ff  0b 00 00 00  08 ff 08 02  00 00 0d ff
0000050: 00 12 00 00  02 ff 02 03  00 00 0b ff  0b 01 00 00
0000060: 08 ff 08 02  00 00 06 ff  06 00 00 00  09 ff 09 05
0000070: 00 00 0b ff  0b 09 00 00  06 ff 06 00  00 00 03 ff
0000080: 03 08 00 00  09 ff 09 0c  00 00 0b ff  0a 0e 00 00
0000090: 0f ff 00 06  00 00 00 ff  00 09 00 00  02 ff 02 0f
00000a0: 00 00 05 ff  05 09 00 00  06 ff 06 09  00 00 06 ff
00000b0: 06 00 00 00  05 ff 05 11  00 00 11 ff  00 0b 00 00
00000c0: 00 ff 00 00  00 00 0a ff  0a 00 00 00  0b ff 0a 0e
00000d0: 00 00 08 ff  08 10 00 00  08 ff 08 02  00 00 0b ff
00000e0: 0b 0b 00 00  00 ff 17 0d  00 00 00 ff  0e 00 00 00
00000f0: 00 ff 0b 00  00 00 06 ff  06 09 00 00  03 ff 03 07
0000100: 00 00 19 ff  00 04 00 00  00 ff 17 02  00 00 10 ff
0000110: 00 0a 00 00  00 ff 00 00  00 00 0e ff  0d 01 00 00
0000120: 00 ff 0f 00  00 00 00 00  00 00 00 00  00 00 00 00
0000130: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
*
0000370: f4 00 f6 09  f2 15 f2 1b  f2 21 f2 27  f0 2d f2 30
0000380: f2 36 f2 3c  f4 42 f2 4b  f0 51 f0 54  f0 57 f2 5a
0000390: f2 60 f2 66  f2 6c 00 00  00 00 00 00  00 00 00 00
00003a0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
*
00003e0:
```

Figure 2.4: FSM implementing DCF, Binary-Byte-Code representation.

# Chapter 3

# WMP details: events, conditions and actions

This section goes into details about fundamental elements of the WMP. As said in the previous chapter, a FSM is built on states, transitions, events, conditions and actions: transitions, that link states one way and are triggered by events or ruled by conditions, eventually start the execution of actions.

For each state the MAC-Engine checks only the events associated to the transitions that exit from that state and waits until one triggers. If a transition is associated only to an event, then it is unique to the given destination state. If instead it is also associated to a condition, then there exists another transition triggered by the same event and associated to the condition negated.

## 3.1 Events

Events are signal generated by the underlying hardware. In the following we report a list of events that can be used to define a state machine and a detailed description for each of them.

- `TX_READY` This event is triggered when the transmitter begins transmitting a packet so that all relevant operations can be handled by the MAC. The basic logic that leads to the transmission of a frame in the current WMP is composed of the following four steps: first, it is checked that a frame is available in the queue, if it is then the hardware is set up for transmitting the frame; a second step chooses when to transmit the frame, e.g., after a SIFS, PIFS, DIFS or after a backoff stage, so that the transmission is scheduled and it will start independently of the WMP. After waiting the scheduled delay (third step), the WMP finalizes hardware setup for the undergoing transmission. It's more important that the event `TX_READY` is triggered when the physical transmitter begins transmitting.

- `TX_IN_PROGRESS` This event is active during the transmission of a frame. There are, in fact, a number of activities that must be done during the transmission of the packet to finalize the transmission itself, e.g., setting up the ack timeout if the frame that is being transmitted requires to be acknowledged. This event is used to help the WMP understanding when these activities, that are handled by action `TX_INFO_UPDATE`, should be actually executed (e.g., check if the transmission was interrupted).

- `TX_10us_ELAPSED` This event is triggered 10us after the end of the current transmission. The WMP architecture, in fact, requires to measure the channel noise after each transmission but this can not be done immediately after otherwise the measure could be affected by the transmission itself. When the event triggers, action `NOISE_MEASUREMENT` is executed.

- `TX_ERROR` This event triggers when an error is detected during transmission and it should be checked by all states involved in transmission activity. If event triggers, action `MANAGE_TX_ERROR` must be executed to reset the transmitter.

- `RX_PLCP` This event triggers when the PLCP of a frame is received. The WMP checks this event to begin handling reception before it terminates, e.g., to schedule the transmission of the acknowledgment if the packet that is being received requires it. If this event triggers, action `RX_PLCP` must be executed.

- `RX_COMPLETE` This event is triggered at the end of the current reception: in this case action `RX_COMPLETE` must be executed.
- `ACK_TIMEOUT` This event triggers when the ack timeout timer expires. The value of the ack timeout is set up during the first stage of transmission if required (e.g., unicast data frame) and accordingly to the code and rate of the packet that is being transmitted. The acknowledgment, in fact, will be transmitted back at the same rate and this allow to correctly choose the ack timeout, that can not be fixed. The timer is started at the end of the current transmission and its expiration means that no ack has been received, otherwise the timer should have been stopped. This event is handled by action `CONTENTION_PARAMS_UPDATE` which, according to the number of transmission attempts will either set up a retry or start the operations to remove the current frame from the queue.
- `RX_ERROR` This event triggers when an error is detected during reception and it should be checked by all states involved in reception activity. If event triggers, action `MANAGE_RX_ERROR` must be executed to reset the receiver.
- `PACKET_IN_TX_QUEUE` This event triggers when the upper layers enqueued a packet in the device queue: it signals to the WMP that the packet is ready for transmission, that will be handled by action `TX_PKT_SCHEDULER`.
- `TIMEOUT_TIMER_0`, `TIMEOUT_TIMER_1` There are two timers that could be used by the user for MAC customization, GPT0 and GPT1. They are activated by a specific action and their timeout is signalled by these pair of events.

## 3.2 Conditions

Conditions are evaluated by checking values reported by hardware registers or by internal software register after they are changed and can be also function of multiple (mixed) values. Conditions are evaluated by the WMP after an event is detected to choose which transition associated to that event should be followed and which action executed, leading to a given final state. This is also the main difference between an event and a condition: the former is considered only when it is raised by the hardware, the latter is always considered to decide which transition should be followed. In the following we report a list of the conditions that can be used to define a state machine and a detailed description of each of them.

- `RX_PACKET == MY_BEACON` This condition evaluates true if the last received frame was a beacon and was transmitted by the associated AP.
- `TX_PACKET == GOOD` This condition is evaluated at the end of a transmission and set up according to a number of checks that determine if the transmission was correctly handled. This condition is used after event `PACKET_IN_TX_QUEUE` triggers.
- `NEED_SEND_ACK` This condition evaluates true if the received frame must be acknowledged: in this case the WMP drives the state machine in a state that will transmit the ack.
- `NEED_WAIT_ACK` This condition evaluates true if the transmitted frame requires an acknowledgment: in this case the WMP must drive the state machine in a “waiting ack” state, which comprises a waiting stage followed by ack reception.
- `BK_VAL != 0` This condition evaluates true if the backoff counter is not null. This condition is used if during the backoff a packet has been received and the backoff was freezed: in this situation the condition is used at the end of the current reception to understand that is need returned in the backoff state to continue the backoff countdown. In the other case, if the backoff value is zero the WMP must return in the idle state.
- `TX_DST_ADDR == PARAM_TX_DST_ADDR_`$n$ This condition evaluates true is the destination MAC address of the packet being transmitted corresponds to the one specified by the nth-state parameter of the condition `PARAM_TX_DST_ADDR_`$n$, $n \in [1, 2, 3]$.
- `RX_SRC_ADDR == PARAM_RX_SRC_ADDR_`$n$ This condition evaluates true is the source MAC address of the packet being received corresponds to the one specified by the nth-state parameter of the condition `PARAM_RX_SRC_ADDR_`$n$, $n \in [1, 2, 3]$.
- `TIMER_`$n$ `== ON` This condition evaluates true if timer GPT$n$ is running, $n \in [0, 1]$.

- `CUR_CHAN == PARAM_CHECK_CHANNEL` This condition evaluates true if the current channel is equal to that specified in the state parameter of the condition whose name is `PARAM_CHECK_CHANNEL`.
- `RX_PACKET == ACK` This condition evaluates true if the last received frame was an acknowledgment. This condition is typically used during ack waiting to detect if something different has been received in place of the expected ack or if the transmitted frame has been correctly acknowledged.
- `TX_SLOTTED` When used, this condition evaluates true periodically: the time period must be set up in the state parameter `PARAM_TIME_SLOT` corresponding to this condition and the timer used by the WMP for evaluating this condition is the internal clock reference; this means that for MACs that maintain synchronization among stations the occurrence of the condition is distributed.

## 3.3 Actions

Actions are elementary operations that can be executed during a state transition to implement complex MAC. In the following we report a list of the actions that can be used to define a state machine and a detailed description of each of them.

- `TX_PACKET` When executed, it finalizes the transmission of the frame that has been previously checked in the queue, analyzed and scheduled for transmission after a selected delay, operations triggered by event `TX_READY`.
- `TX_INFO_UPDATE` This action is executed after a transmission has started and it prepares the environment to handle the operations that will be executed after transmission end. The action is usually run during packet transmission and is triggered by event `TX_IN_PROGRESS`.
- `NOISE_MEASUREMENT` When executed, it handled all operations needed to measure the channel noise. This task must be executed 10us after each transmission and for this reason it is triggered by event `TX_10us_ELAPSED`.
- `MANAGE_TX_ERROR` When executed, it handles errors that have been detected during transmission. It is triggered by event `TX_ERROR`.
- `RX_PLCP` The receiver loop is divided into two stages. The first stage starts when the PLCP of an incoming packet has been decoded correctly and it is triggered by event `RX_PLCP`. The WMP configures the hardware for finalizing the frame reception (or if needed to stop it, discarding the bytes that have been already received). In this stage the WMP may also prepare the acknowledgment if the incoming packet requires it, e.g., by setting up the code and the rate and by scheduling it after a SIFS after the reception will be concluded. The second stage is triggered at the end of the reception (see below).
- `RX_COMPLETE` It must be executed after event `RX_COMPLETE`: the WMP concludes the reception of the frame (and the receiver loop as well).
- `MANAGE_RX_ERROR` When executed, it handles errors that have been detected during frame reception and that triggered event `RX_ERROR`.
- `SET_TIMER_`$n$ (`PARAM_TIMER_[`$n$`, `$m$`]`) When executed, it activates timer GPT$n$, $n \in [0, 1]$ using one of the two associated state parameters $m \in [0, 1]$.
- `TX_PKT_SCHEDULER` When executed, it takes care of all operations involved in the second step of the transmission loop, that is after having checked that the current packet in the queue can be transmitted, it schedules the actual transmission by choosing the time it will start with respect to some event in the past (e.g., after a SIFS after the end of the reception of the previous frame). Backoff for transmissions that require it is chosen at this step. This action uses the state parameter `PARAM_BACKOFF` for set the transmission delay and the value of the backoff, in this way:
  - 0xFFFF to keep the legacy behavior with respect to the implemented;
  - 0x0FFF to transmit immediately;
  - 0x00 – 0xFF to set a specific backoff value;

- `REPORT_TX_STATUS_TO_HOST` When executed, it reports to upper layers information about transmission status (e.g., number of attempts, delivery failure etc).
- `SUPPRESS_THIS_TX_FRAME` When executed, it discards the current packet in the transmission queue, e.g., this is executed to remove an outstanding packet because too old. Condition `TX_PACKET == GOOD` controls this action.
- `CHANGE_CHANNEL (PARAM_SET_CHANNEL)` When executed, it modifies the channel by setting it to that specified by state parameters `PARAM_SET_CHANNEL`.
- `RESET_CHANNEL` When executed, it modifies the channel by setting the same used on the associated AP if the current is different.
- `ACTIVATE_TX_DIRECT_LINK` This action affects the MAC addresses of the frame that will be transmitted and on the MAC frame. When executed, the frame header is modified: to better understand the change we first report the original header, then the modified one:
  - **Original header** We consider a frame that a source station SA sends to distribution system DS for forwarding to destination station DA:

| To DS | From DS | Address 1 | Address 2 | Address 3 |
|---|---|---|---|---|
| 1 | 0 | BSSID | SA | DA |

Table 3.1

  - **Modified header** Distribution system DS (the AP) hop is skipped, so addresses are rearranged like follows:

| To DS | From DS | Address 1 | Address 2 | Address 3 |
|---|---|---|---|---|
| 0 | 1 | DA | BSSID | SA |

Table 3.2

  This allows a direct transmission between SA and DA, clearly they must be in their coverage. This action should hence be used when a direct link is activated between two stations and must be used paired with action ACTIVE RX DIRECT LINK on the receiver otherwise the acknowledgment will be handled in the wrong way.

- `ACTIVATE_RX_DIRECT_LINK` When this action is executed, the outstanding ack that will be transmitted to acknowledge the received frame will be modified: in this case the unique address in the ack will be copied from Address 3 of the incoming frame instead of Address 2. This action should be used paired with `ACTIVATE_TX_DIRECT_LINK` to implement a direct link between a pair of stations.
- `CONTENTION_PARAMS_UPDATE_FAIL` When executed, it increases the values of the contention parameters according to the values of the backoff parameters(`PARAM_INFLATION_MUL` and `PARAM_INFLATION_ADD` ). This action should be executed when the reply frame (i.e., ACK frame) for the last transmitted packet was not received within the given timeout. When the maximum number of transmission attempts has been reached, upper layers will be reported about that.
- `CONTENTION_PARAMS_UPDATE_SUCCESS` When executed, it decreases (or set to zero) the values of the contention parameters according to the values of the backoff parameters(`PARAM_DEFLATION_DIV` and `PARAM_DEFLATION_SUB` ). This action should be executed when the reply frame (i.e., ACK frame) for the last transmitted packet was received within the given timeout. If the new value of the current contention window goes below the allowed minimum, then it is reset to the allowed minimum.
- `TX_PACKET_WITHOUT_RECEIVE_ACK` When executed, it finalizes the transmission of the frame that has been previously checked in the queue, analyzed and scheduled for being transmitted without waiting for any reply frame (i.e., ACK frame). This action is triggered by event `TX_READY`.

- `RX_COMPLETE_WITHOUT_SEND_ACK` It can be executed after event `RX_COMPLETE`: the WMP concludes the reception of the frame without scheduling any reply frame (i.e., ACK frame).
- `RESET_ACK_TIMEOUT` When executed, the action reset the ack timeout condition.

## 3.4 State parameters

In the following we report a short description of all state parameters:

`BOOTSTRAP PARAMS`

- `PARAM_STATE_MACHINE_START` This is a bootstrap parameter, it defines the initial state of the FSM after Byte-Code starts. Switches between Byte-Codes can happen only if the FSM of the current Byte-Code is in its initial state. This requisite guarantees that when a Byte-Code is deactivated, there are no pending operations in the WMP.
- `PARAM_CHANNEL_MACLET` This is a bootstrap parameter, defines the initial channel of the radio for the designed Byte-Code.
- `PARAM_CW_MIN` This is a bootstrap parameter, defines the initial value of MIN CONTENTION WINDOWS for the designed Byte-Code.
- `PARAM_CW_MAX` This is a bootstrap parameter, defines the initial value of MAX CONTENTION WINDOWS for the designed Byte-Code.
- `PARAM_CW_CUR` This is a bootstrap parameter, defines the initial value of CUR CONTENTION WINDOWS for the designed Byte-Code.

`ENHANCED PARAMS`

- `PARAM_BACKOFF` This parameter defines the medium access time for frame transmission. It is used by action `TX_PKT_SCHEDULER` that is executed after a frame is in the tx queue to set transmission parameters like the interframe space (SIFS, PIFS, DIFS), and the backoff value that will be used to wait before transmission. Backoff will be set up according to the following values
  - 0xFFFF: standard, contention window value begins at a minimum and doubles at every collision, till a maximum value is reached;
  - 0x0FFF: frame is transmitted immediately after action execution, there could be some delay due to code execution time;
  - 0x00 –0xFF: in this case the parameter sets the exact value of the backoff that will be used.
- `PARAM_SET_CHANNEL` This parameter defines the channel used to tune the radio when action `CHANGE_CHANNEL` is executed.
- `PARAM_TX_DST_ADDR_`$n$ This parameter specifies one 48bit MAC address that is compared in condition `TX_DST_ADDR == PARAM_TX_DST_ADDR_`$n$ with the destination address of the frame that will be transmitted, $n \in [1, 2, 3]$.
- `PARAM_RX_SRC_ADDR_`$n$ This parameter specifies one 48bit MAC address that is compared in condition `RX_SRC_ADDR == PARAM_RX_SRC_ADDR_`$n$ with the source address of the frame that is being received.
- `PARAM_GPT_`$n$`_`$m$`_CNTHI` , `PARAM_GPT_`$n$`_`$m$`_CNTHI` These two 16bit parameters line up in their corresponding 32bit timestamp parameter `PARAM_TIMER_[`$n$`, `$m$`]` to setup the initial value of timer GPT$n$ by action `SET_TIMER_`$n$, $n \in [0, 1]$, $m \in [0, 1]$. Given that $m$ can assume two different values, two different timestamps can be built for each timer (GTP0 and GPT1).
- `PARAM_CHECK_CHANNEL` This parameter is used in condition `CUR_CHAN == PARAM_CHECK_CHANNEL` to verify that the current radio channel is that in the parameter.
- `PARAM_TIME_SLOT` This parameter is used in condition `TX_SLOTTED` to compute the time slot for next transmission. It is fundamental to implementation of TDM based MACs.

`BACKOFF PARAMS`

- `PARAM_INFLATION_MUL` This is a backoff parameter, defines how to increase the value of the current contention window.
- `PARAM_INFLATION_ADD` This is a backoff parameter, defines how to increase the value of the current contention window.

Parameters `PARAM_INFLATION_MUL` and `PARAM_INFLATION_ADD` defines the function that is used to update the current contention window, that is

cwcur = 2 * `PARAM_INFLATION_MUL` + `PARAM_INFLATION_ADD`

- `PARAM_DEFLATION_DIV` This is a backoff parameter, defines how to decrease the value of the current contention window.
- `PARAM_DEFLATION_SUB` This is a backoff parameter, defines how to decrease the value of the current contention window.

Parameters `PARAM_DEFLATION_DIV` and `PARAM_DEFLATION_SUB` defines the function that is used to update the current contention window, that is

cwcur = 2 / `PARAM_INFLATION_MUL` - `PARAM_INFLATION_ADD`

# Chapter 4

# WMP-Editor

## 4.1 Overview

**W**ireless **M**AC **P**rocessor Graphic **E**ditor (WMP-Editor) is a graphical tool that represents state machine programs as transition graphs. Users can edit WMP graphically, adding new states and transitions and customizing the WMP behavior working on its atomic elements, namely conditions, actions and events as introduced in Chapter 3. The same tool can be used as a compiler to translate the transition graph into a Byte-Code.

In this chapter we describe the WMP-Editor introducing the different design styles supported by the tool.

## 4.2 Features

The WMP-Editor renders a state machine into a user-friendly graphical representation. Thanks to the editor, the programmer can design a MAC program without having to care at Byte-Code labels for coding events, actions and conditions. It is up to the tool to translate the graphical representation of the designed state machine into a low-level Byte-Code table that can be interpreted by the MAC-Engine.

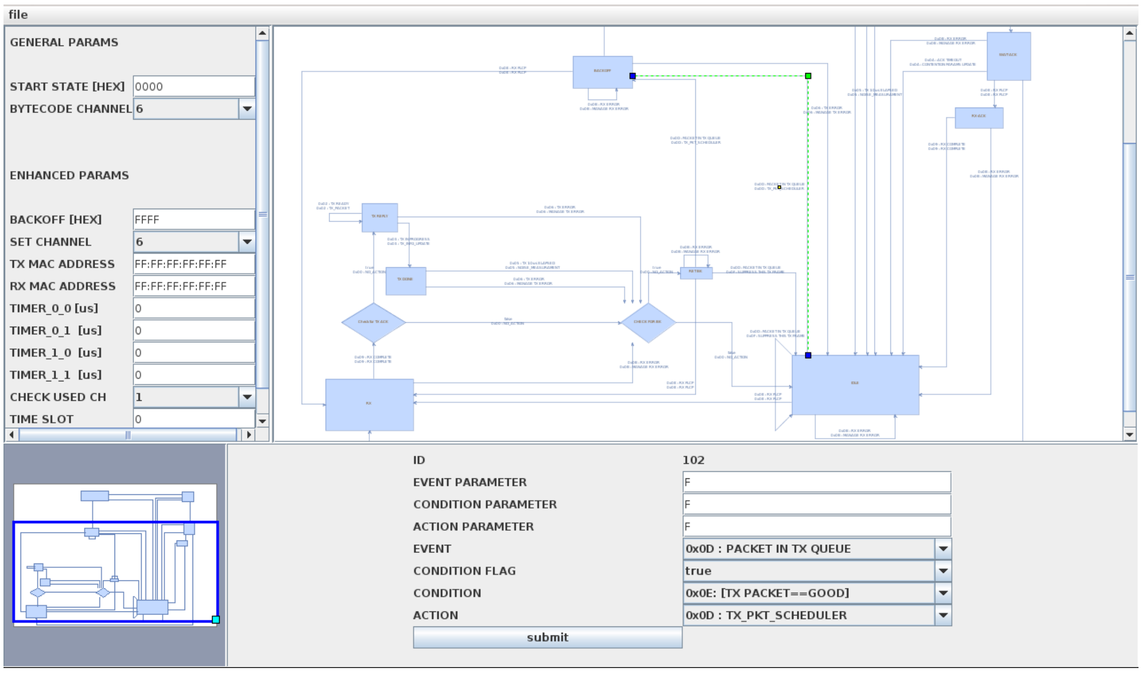
There are two main types of editor elements: **Blocks** and **Transitions**. Blocks are graphical boxes representing states of the MAC-Program, while transitions are graphical arrows representing *state changes*. Each block (i.e. each state) has a number of outgoing transitions triggered by the occurrence of events and enabled by the verification of an optional condition.

Since the MAC-Engine implemented on the commercial AirForce One card by Broadcom is not able to detect interrupt signals (it reveals events only by means of a periodic polling of some event registers), in our implementation there is not a conceptual difference between events and conditions. This means that the graphical compiler always maps a transition triggered by event $e$ and enabled by the verification of condition $c$ into a sequence of two transitions: i) an asynchronous one, triggered by transition event $e$, as soon as the Engine reveals the occurrence of that event; ii) a synchronous one, immediately triggered after the first transition towards two possible states according to the TRUE/FALSE value of condition $c$. The first transition is performed towards an intermediate state (whose permanence time is theoretically zero), from which only two outgoing transitions (corresponding to the RRUE/FALSE value of the condition) are possible. Although the programmer could completely neglect these implementation details and define state transitions by specifying both the triggering event and the enabling condition, the tool also allows to explicitly deal with the intermediate states required for verifying the enabling conditions. In this case, normal transitions are defined by means of a triggering event only, and condition states are added in the graphical representation of the machine. For improving the machine readability, conditions states are blocks with a different shape, that gives emphasis on the flow splitting into two different possible outcomes (i.e. TRUE/FALSE).

In section 4.3.1 we describe how to effectively use the two different programming styles for customizing the machine behavior.

## 4.3 Editor Description

The basic Layout of WMP-Editor (see Figure 4.1) is an all-in-one window organized into three main frames: the left most frame, containing the global parameters of the state machine; the middle frame where the graphical state machine is composed; and the bottom frame that hosts the user interface for creating and modifying program states and transitions. In details:



GENERAL PARAMS

| | |
|---|---|
| START STATE [HEX] | 0000 |
| BYTECODE CHANNEL | 6 |

ENHANCED PARAMS

| | |
|---|---|
| BACKOFF [HEX] | FFFF |
| SET CHANNEL | 6 |
| TX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| RX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| TIMER_0_0 [us] | 0 |
| TIMER_0_1 [us] | 0 |
| TIMER_1_0 [us] | 0 |
| TIMER_1_1 [us] | 0 |
| CHECK USED CH | 1 |
| TIME SLOT | 0 |

| | |
|---|---|
| ID | 102 |
| EVENT PARAMETER | F |
| CONDITION PARAMETER | F |
| ACTION PARAMETER | F |
| EVENT | 0x0D : PACKET IN TX QUEUE |
| CONDITION FLAG | true |
| CONDITION | 0x0E: [TX PACKET==GOOD] |
| ACTION | 0x0D : TX_PKT_SCHEDULER |

Figure 4.1: WMP-Editor Layout

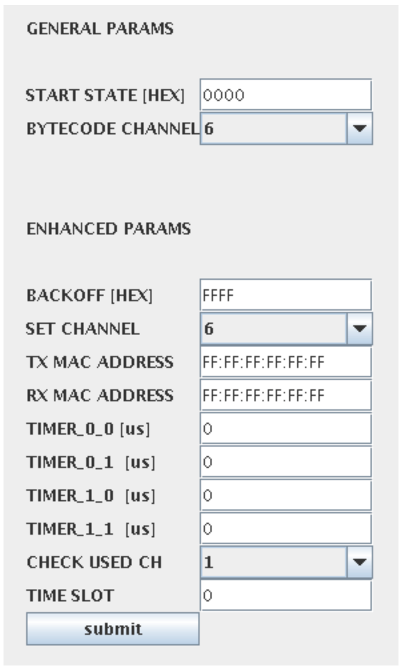
- **Parameters Frame** It includes all the environment variables of the state machine. In Figure 4.2(a) we can distinguish two types of parameters: **General** and **Enhanced** parameters. General Parameters set the value of the WMP configuration registers (e.g. the hardware register specifying the operating channel) and the initial state from which the MAC-Engine starts the execution. The Enhanced Parameters allow to specify other program parameters, not strictly related to the default configuration registers, such as MAC addresses to be used for filtering purposes, a channel hopping sequence, a time slot interval, a pre-defined constant backoff value, and so on.

- **Machine Building Frame** It displays the state blocks and the transitions defined by the programmer. WMP-Editor uses a simple right-click pop-up to add and edit state blocks and transitions, as shown in Figure 4.2(b).

- **User API Frame** It is the bottom area of WMP-Editor where programmers modify the properties of state blocks, condition blocks (if explicitly included in the machine representation), and transition elements, by specifying events, conditions and actions for each transition from the set of available API.

### 4.3.1 WMP Machines

A MAC program is defined in terms of an extended state machine, i.e. a state machine in which transitions triggered by a given event can be enabled by the verification of a logical condition. An extended state machine allows to reduce the state space, since it decouples the actual state of the program into a "protocol" state (explicitly represented in the transition graph) and a "configuration" state (i.e. a list of registers), on which conditions are verified. Different approaches can be used for implementing a state machine, according to the sequence of condition verifications and action executions. In a Mealy state machine, the action is executed only if the transition is activated (i.e. the condition is verified after the occurrence of the transition event). In other cases, it could be useful to perform the action before the verification of the condition. This operation can still be mapped into a Mealy state in which (as anticipated before) a first transition without an enabling condition is performed towards an intermediate state. Before entering the new state, called condition state, the action is performed, while after entering the new state a new transition is immediately started (the trigger event is a null event) for verifying the condition and moving to the final state.



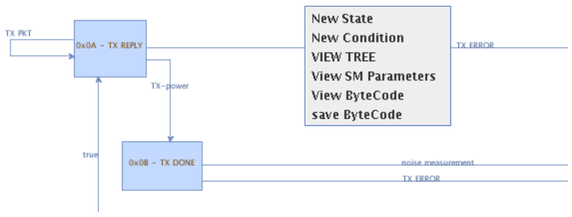
GENERAL PARAMS

|  |  |
|---|---|
| START STATE [HEX] | 0000 |
| BYTECODE CHANNEL | 6 |

ENHANCED PARAMS

|  |  |
|---|---|
| BACKOFF [HEX] | FFFF |
| SET CHANNEL | 6 |
| TX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| RX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| TIMER_0_0 [us] | 0 |
| TIMER_0_1 [us] | 0 |
| TIMER_1_0 [us] | 0 |
| TIMER_1_1 [us] | 0 |
| CHECK USED CH | 1 |
| TIME SLOT | 0 |

(a)

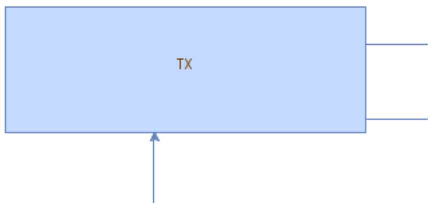


(b)

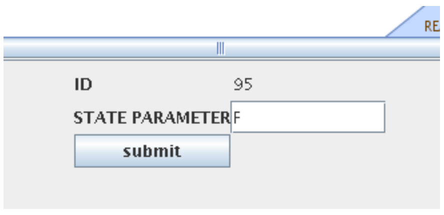
Figure 4.2: Ambient parameters 4.2(a), Pop-Up Menu 4.2(b)

For defining a state machine in the graphical editor, it is enough to start with the definition of the states. A new state can be added in the machine building frame by means of the pop-up menu, while a state label can be added by operating on the user API frame. Figure 4.3 shows a simple state example.

Condition states can be created as normal states, with the only different that there are only two outgoing transitions linked to the same condition verification (TRUE/FALSE value). The condition to be verified from this state is specified in the state definition (in terms of condition label, selected from the available API). Such a state can also improve the machine readability, since it works as an IF statement in an imperative programming language. States (normal states and condition states) are connected by transitions. Transitions from normal states can specify events, conditions and actions or simply events and actions in absence of enabling conditions. Transitions from condition states specify only the TRUE/FALSE flag of the condition and the action. Figure



(a) State

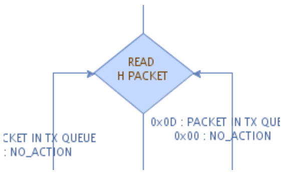


|  |  |
|---|---|
| ID | 95 |
| STATE PARAMETER | F |

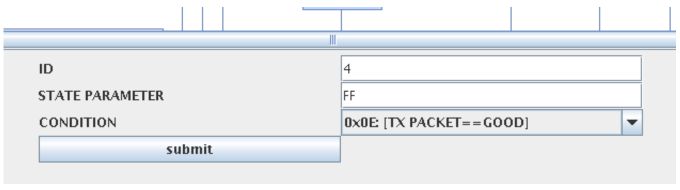
(b) State Parameters

Figure 4.3: State

4.5 shows three different examples of transitions: **event-triggered transitions** (Figure 4.5(a)), **condition-verifying transitions** (Figure 4.5(b)) and **event-triggered transitions with enabling conditions** (Figure 4.5(c)). The choice of a specific type of transition may depend on the programmer style, but in many cases may be optimized for reducing the number of states or the number of transitions. For example, the usage of an explicit condition state to be verified in multiple transitions (let $n$ be the number of these transitions) for entering the same states may reduce the number of transitions from $2 \cdot n$ (two transitions for each condition outcome) to $n+2$ ($n$ transitions without enabling conditions and two more transitions from the condition state). Conversely, if we need to verify multiple conditions from a given state, it can be more efficient to use event-triggered condition-verifying conditions for limiting the state space. Moreover, it is possible



(a) Condition



ID 4
STATE PARAMETER FF
CONDITION 0x0E: [TX PACKET==GOOD]

(b) Condition Parameters

Figure 4.4: Condition

to use this type of transmissions for combining the verification of two simultaneous conditions. Each type of transition can be configured by specifying a sub-set of parameters offered by the user API frame.

**Event-triggered transitions** Figure 4.5(a) shows an event-triggered transition. In this example, the current state is a `BACKOFF` state, from which different events are monitored including event `TX_READY`. When the WMP reveals such an event (by polling the corresponding event register) a transition is immediately performed towards the TX state. The transition fields to be specified for this type of transitions are:

1. **EVENT PARAMETER**: an optional parameter for defining a parametrized event (4 bits);
2. **ACTION PARAMETER**: an optional parameter to be passed to the transition action (4 bits);
3. **EVENT**: the transition event (selected from the available API) in terms of event label;
4. **ACTION**: the action to be performed before entering the new state (selected from the available API) in terms of action label.

NOTE: Normal states might have multiple transitions of this type.

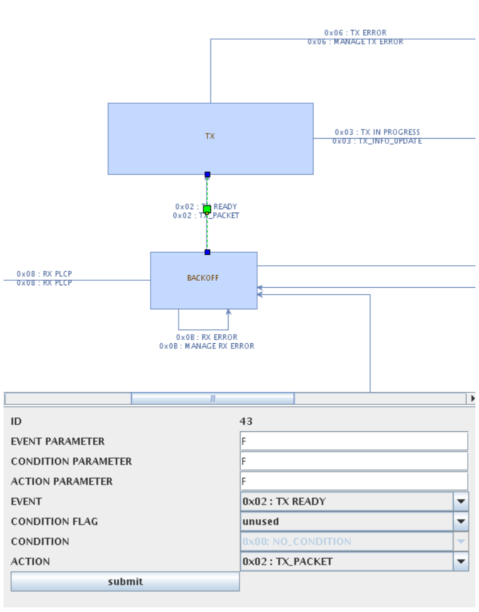
**Condition-verifying transitions** Figure 4.5(b) shows an example of condition-verifying transition. This type of transitions are configured differently from the event-triggered transitions. Specifically, it is required to specify the following fields only:

1. **ACTION PARAMETER**: an optional parameter to be passed to the transition action (4 bits);
2. **CONDITION FLAG**: the condition outcome, i.e. the TRUE or FALSE state of the condition register linked to the condition state;
3. **ACTION**: the action to be performed before entering the new state (selected from the available API) in terms of action label.

NOTE: Condition State accept ONLY TWO transitions.

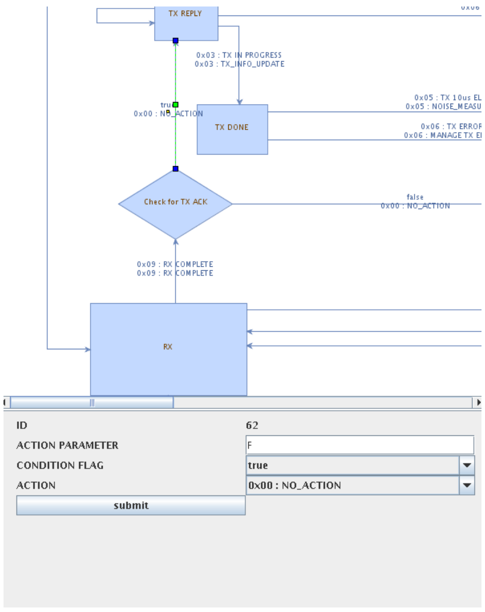
**Event-triggered transitions with enabling conditions** This transition type corresponds to the transition type used in Mealy extended state machines. It is formally equivalent to an event-triggered transition, in which a condition is specified for enabling or not the transition event. In other words, after that the WMP reveals the occurrence of the transition event, a condition is verified before performing the state transition. The configuration of this type of transition requires to specify:

1. **EVENT PARAMETER**: an optional parameter for defining a parametrized event (4 bits);
2. **CONDITION PARAMETER**: an optional parameter for defining a parametrized condition (4 bits);
3. **ACTION PARAMETER**: an optional parameter to be passed to the transition action (4 bits);
4. **EVENT**: the transition event (selected from the available API) in terms of event label;



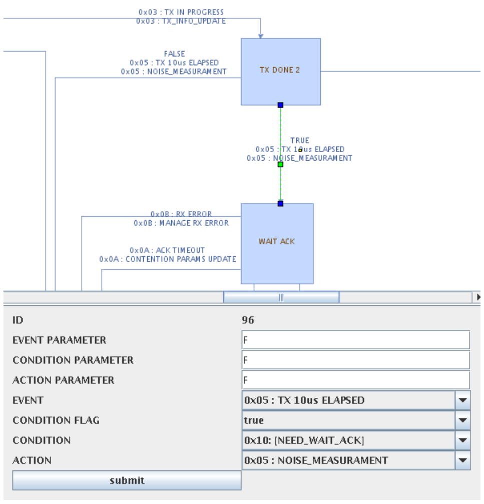
| | |
|---|---|
| ID | 43 |
| EVENT PARAMETER | F |
| CONDITION PARAMETER | F |
| ACTION PARAMETER | F |
| EVENT | 0x02 : TX READY |
| CONDITION FLAG | unused |
| CONDITION | 0x00: NO_CONDITION |
| ACTION | 0x02 : TX_PACKET |

(a) Event State Transition



| | |
|---|---|
| ID | 62 |
| ACTION PARAMETER | F |
| CONDITION FLAG | true |
| ACTION | 0x00 : NO_ACTION |

(b) Condition Transition



| | |
|---|---|
| ID | 96 |
| EVENT PARAMETER | F |
| CONDITION PARAMETER | F |
| ACTION PARAMETER | F |
| EVENT | 0x05 : TX 10us ELAPSED |
| CONDITION FLAG | true |
| CONDITION | 0x10: [NEED_WAIT_ACK] |
| ACTION | 0x05 : NOISE_MEASURAMENT |

(c) Condition Transition Nested

Figure 4.5: Parameters of Transition

5. **CONDITION FLAG**: tristate field: UNUSED/TRUE/FALSE, used in Condition transitions and Event-Condition nested Transitions.
6. **CONDITION**: Unused in Event Transition identifier
7. **ACTION**: the action to be performed before entering the new state (selected from the available API) in terms of action label.

Chapter 5

# MAC Design

## 5.1 Introduction

This chapter describes three examples of state machine graphical implementation, namely the Distributed Coordination Function (DCF), the Time Division Multiple Access (TDMA) and the Direct Link setup (DLS). The chapter is organized like an HowTo for helping users understanding how to design a MAC using the WMP Editor. The following paragraphs explain the logic used to create a state machine and the implementation issues that need to be solved to obtain working state machines and Byte-Codes, in particular:

- **DCF** This state machine implements basic variants of the standard Distributed Coordination Function (DCF): here the Backoff is customized according to three evolutions, normal, fixed and disabled.
- **TDM** (or Pseudo-TDMA) This state machine does not update the contention windows parameter and schedules packet transmissions at fixed slot times.
- **Direct Link Setup** This state machine is derived from DCF and changes the packet forwarding style according to the MAC address of the destination: for a set of selected targets the standard transmission procedure (each packet is sent to the AP) is overridden and packets are sent directly to destination stations without forwarding through the Access Point.

## 5.2 Distributed Coordination Functions (DCF)

The DCF state machine implements standard IEEE 802.11 functions using WMP APIs. Figure 5.1 shows the graph associated to the state machine that can be logically split into two parts: one handling incoming packets reported on the left side of the Figure, another handling outgoing packets and reported on the right side. Whether switching to one or the other is decided by the initial state `IDLE` according to the following events:

- `PACKET_IN_TX_QUEUE` - Start Transmit operation mode;
- `RX_PLCP` - Start Transmit operation mode;
- `RX_ERROR` - Manage error.

Event `RX_ERROR` is triggered by the receiver if an invalid packet is received, either because the receiver is not able to finish the reception of the current packet or because a corrupted PLCP is detected. The corresponding action `MANAGE_RX_ERROR` resets the receiver. It is worth noting that this event must be checked in all states that control reception. We describe Receive and Transmit operation modes respectively in Section 5.2.1 and 5.2.2.

### 5.2.1 Reception operation mode

When in state `RX`, two events may be triggered:

- `RX_COMPLETE`
- `RX_ERROR`

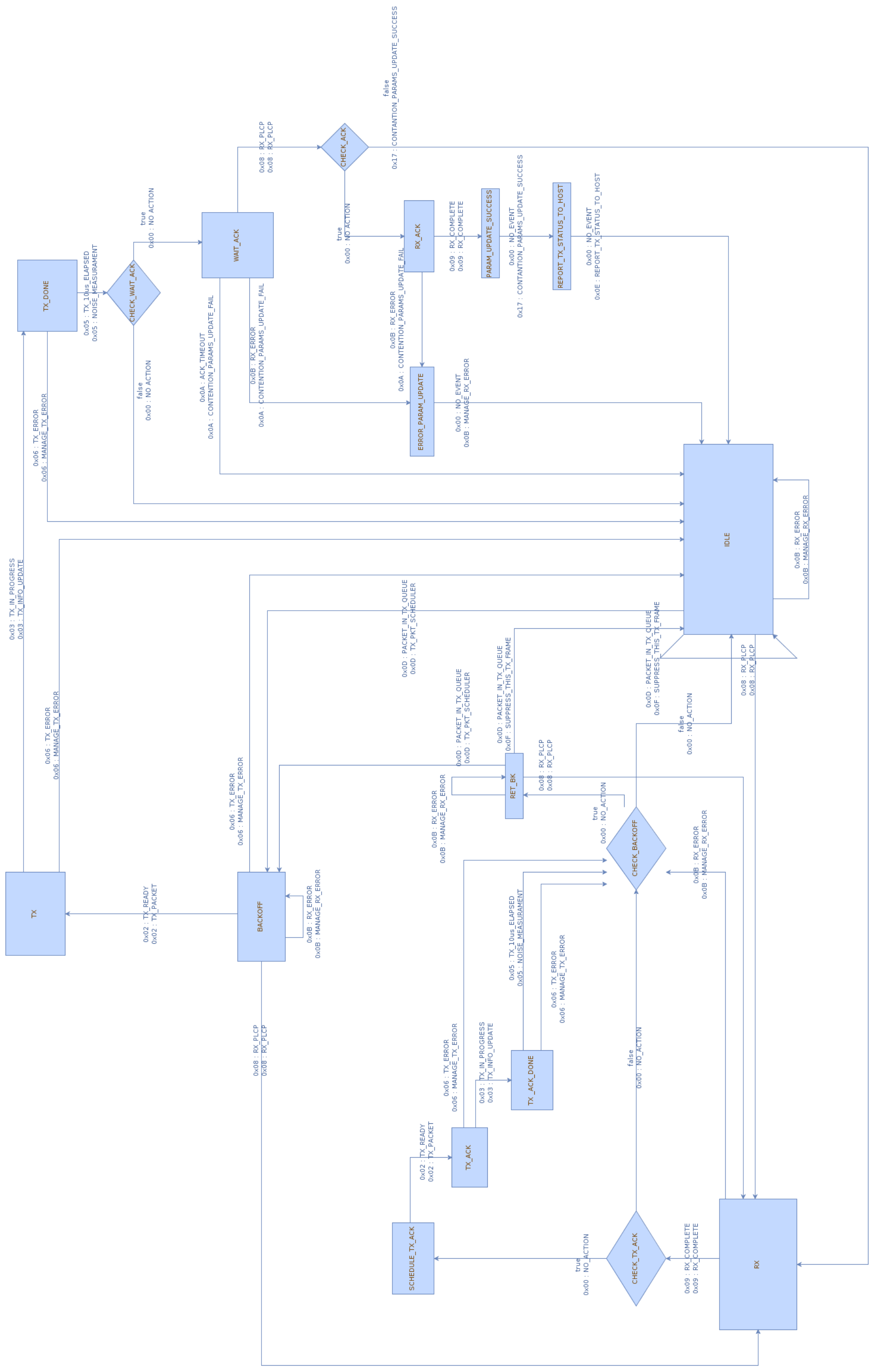


Figure 5.1: DCF

Event RX_COMPLETE indicates that the incoming frame ended and moves the State Machine to conditional state CHECK_TX_ACK that checks condition NEED_SEND_ACK to determine if the received frame is to be acknowledged or not.

#### 5.2.1.1 Don't ack frame

When the received frame does not need ACK reply, the state machine evolves to conditional state CHECK_BACKOFF that checks the backoff value: if backoff is null, then state machine moves to IDLE state, otherwise to conditional state CHECK_BACKOFF.

#### 5.2.1.2 Do ack frame

When the received frame needs ACK reply, the state machine moves to state SCHEDULE_ACK where it will wait for two events:

- TX_READY, transition executes action TX_PACKET and moves the state machine to state TX_ACK;
- TX_ERROR, transition manages transmission errors by executing action MANAGE_TX_ERROR and moves the state machine to state CHECK_BACKOFF.

For the sake of clarity event TX_ERROR should be checked by all states that manage frame transmission: in the following we will not mention that event explicitly anymore.

When in state TX_ACK, the state machine waits event TX_IN_PROGRESS, executes action TX_INFO_UPDATE and evolves to state TX_ACK_DONE.

In state TX_ACK_DONE, the state machine waits event TX_10us_ELAPSED, executes action NOISE_MEASUREMENT to get a noise sample after transmission and evolves to state CHECK_BACKOFF that is used to schedule the transmission of a frame when there is one in the transmission queue.

When conditional state CHECK_BACKOFF is reached, state machine checks condition BK_VAL != 0: if true evolves to state RET_BK without executing any action, otherwise state machine returns to state IDLE. State RET_BK waits for two events:

- RX_PLCP, transition executes action RX_PLCP and goes to state BACKOFF
- PACKET_IN_TX_QUEUE, transition leads the MAC-Engine to check condition TX_PACKET == GOOD: if verified then state machine evolves to state BACKOFF after executing action TX_PKT_SCHEDULER. If instead the condition is not verified, it evolves to state IDLE after executing action
  SUPPRESS_THIS_TX_FRAME.

### 5.2.2 Transmit operation mode

state machine switches to this operation mode from state IDLE when it detects event PACKET_IN_TX_QUEUE. In this case the MAC-Engine checks condition TX_PACKET == GOOD: if verified then state machine evolves to state BACKOFF after executing action TX_PKT_SCHEDULER. If instead the condition is not verified, it evolves to state IDLE after executing action SUPPRESS_THIS_TX_FRAME.

State BACKOFF is characterized by four outgoing transitions, that are selected whenever one of the following events verify:

- RX_PLCP - triggers evolution to state RX for handling frame arrival during backoff;
- TX_READY - executes action TX_PACKET and goes to state TX;
- TX_ERROR - to manage transmission errors and goes to state IDLE;
- RX_ERROR - to manage errors in the receiver during backoff, selfloop.

When in state TX, the state machine waits event TX_IN_PROGRESS, executes action TX_INFO_UPDATE and evolves to state TX_DONE. Here event TX_10us_ELAPSED triggers checking condition NEED_WAIT_ACK: if verified state machine evolves to state WAIT_ACK otherwise to state IDLE. In both cases action NOISE_MEASUREMENT is executed.

State WAIT_ACK checks three events:

- RX_PLCP - if verified state machine executes action RX_PLCP and evolves to conditional state CHECK_ACK; otherwise to state RX;

- ACK_TIMEOUT - this event is raised when the timeout set for waiting the ACK reply expires, if verified then state machine executes action CONTENTION_PARAMS_UPDATE_FAIL and state machine evolves to state IDLE;
- RX_ERROR - this event is raised if some error occurs during the reception of the ACK, if this event is verified the state machine executes action CONTENTION_PARAMS_UPDATE_FAIL and evolves to state ERROR_PARAM_UPDATE.

Conditional state CHECK_ACK checks RX_PACKET == ACK: if true state machine evolves without actions to state RX_ACK otherwise goes to state RX.

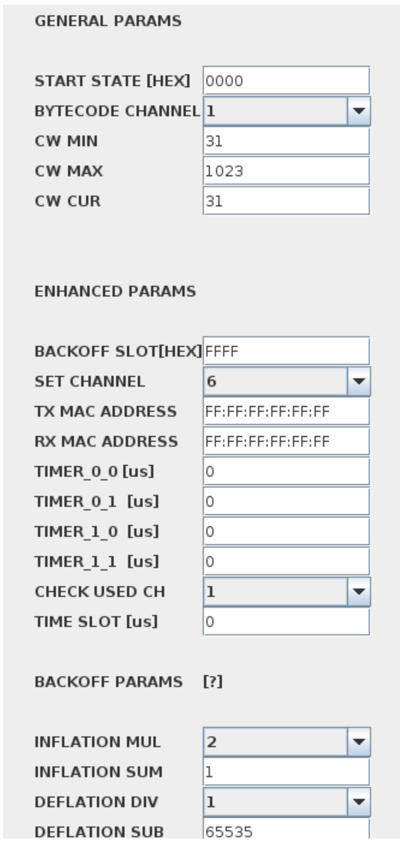
State RX_ACK waits for two events:

- RX_COMPLETE that eventually execute action RX_COMPLETE and evolves state machine to state PARAM_UPDATE_SUCCESS;
- RX_ERROR - this event is raised if some error occurs during the reception of the ACK, if this event is verified the state machine executes action CONTENTION_PARAMS_UPDATE_FAIL and evolves to state ERROR_PARAM_UPDATE.

In state PARAM_UPDATE_SUCCESS state machine checks no events, it executes action CONTENTION_PARAMS_UPDATE_SUCCESS and goes to state REPORT_TX_STATUS_TO_HOST. In last state REPORT_TX_STATUS_TO_HOST state machine checks no events, it executes action REPORT_TX_STATUS_TO_HOST and returns to state IDLE.

### 5.2.3 State Parameters in DCF

Figure 5.2 reports the state parameters that build the configuration of the state machine. Since this implementation is built on standard APIs, the Enahanched Parameters are not used.



GENERAL PARAMS

| | |
|---|---|
| START STATE [HEX] | 0000 |
| BYTECODE CHANNEL | 1 |
| CW MIN | 31 |
| CW MAX | 1023 |
| CW CUR | 31 |

ENHANCED PARAMS

| | |
|---|---|
| BACKOFF SLOT[HEX] | FFFF |
| SET CHANNEL | 6 |
| TX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| RX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| TIMER_0_0 [us] | 0 |
| TIMER_0_1 [us] | 0 |
| TIMER_1_0 [us] | 0 |
| TIMER_1_1 [us] | 0 |
| CHECK USED CH | 1 |
| TIME SLOT [us] | 0 |

BACKOFF PARAMS [?]

| | |
|---|---|
| INFLATION MUL | 2 |
| INFLATION SUM | 1 |
| DEFLATION DIV | 1 |
| DEFLATION SUB | 65535 |

Figure 5.2: State parameters in DCF configuration

### 5.2.4 DCF programmability

Action TX_PKT_SCHEDULER uses parameter PARAM_BACKOFF. By playing with this parameter it is possible to change the 802.11 backoff rule by setting specific values as follows:

- DEFAULT mode: 0xFFFF, i.e. standard exponential backoff rule;
- NO BACKOFF TRANSMISSION mode: 0x0FFF, in this case transmit a packet without doing backoff;
- FIXED BACKOFF VALUE mode: a value in the range [0x00, 0xFF] is used as backoff.

## 5.3 Time Division Multiple Access (TDMA)

Figure 5.3 shows the TDMA implementation. This is a straightforward modification of the DCF State Machine: a new transition from state `IDLE` is triggered by event `PACKET_IN_TX_QUEUE` which eventually evolves the state machine to conditional state `CHECK_TDM_SLOT`. Here if condition `TX_SLOTTED` is true, state machine evolves to conditional state `CHECK_TX_PACKET_GOOD` otherwise returns to state `IDLE`. In conditional state `CHECK_TX_PACKET_GOOD` condition `TX_PACKET == GOOD` is checked: if true state machine executes action `TX_PKT_SCHEDULER` and goes to state `BACKOFF` otherwise it executes action `SUPPRESS_THIS_TX_FRAME` and returns to state `IDLE`.

### 5.3.1 State Parameter in TDMA

The main parameters are `PARAM_BACKOFF` and `PARAM_TIME_SLOT`. The former is set to `0x0FFF`, the latter is the value in microseconds of the time slot for TDMA. Figure 5.4 shows a snapshot of configuration parameters for TDMA state machine:

## 5.4 Direct Link

Direct Link (DL) is a variation of DCF that allows two stations to establish a direct connection bypassing the Access Point for interstation frames. DL is implemented in two variants: **Direct Link Setup (DLS)** and **Direct Channel Link Setup (DCLS)**. Both variants modify MAC header to transmit the frame addressed to the target station.

### 5.4.1 DLS

transmits direct frames using the same radio channel of the AP. There are no synchronization problem because beacons are received as usual. Since this is a demo state machine, only one direct target is allowed and, in fact, it is encoded in the Byte-Code settings as a target MAC address (of the direct link station). When a DLS station sends a frame to its direct link target, the MAC-Engine changes the MAC header of the frame. As shown in Figure 5.5, Direct Link Setup needs two additional conditional states respect to standard DCF, one before state `BACKOFF` and another one before state `RX`. First change is near state `BACKOFF` that is reached from states `IDLE` or `RET_BK`: in DLS state machine, before reaching state `BACKOFF` a new state called `CHECK_TX_ADDR` must be visited. In this state state machine checks condition `TX_DST_ADDR == PARAM_TX_DST_ADDR_0` : in both cases state machine evolves to state `BACKOFF` but if condition is true MAC-Engine executes action `ACTIVATE_TX_DIRECT_LINK`.

Second change is in state `RX` that is reached from states `IDLE`, `BACKOFF` and `WAIT_ACK`. DLS state machine implementation needs an additional state called `CHECK_RX_ADDR`. Here the state machine checks condition `RX_SRC_ADDR == PARAM_RX_SRC_ADDR_0` : in both cases state machine evolves to state `RX` but if condition is true MAC-Engine executes action `ACTIVATE_RX_DIRECT_LINK`.

### 5.4.2 DCLS

transmits direct frames on a separate radio channel. This improves the throughput performance even if a channel hopping mechanism is needed to periodically switch back to the AP radio channel for receiving the beacons and avoid synchronization issues. As shown in Figure 5.6 the DCLS implementation slightly differs from that of the DLS. Stations in a BSS, in fact, synchronize their internal clock with that of the AP using the timestamp inside each received beacon. For this reason it is necessary to periodically switch the channel of the station back to that of the AP. New states provides periodical or per-event channel hopping. In RX section we introduce a block that checks condition `RX_PACKET == MY_BEACON` to verify if the frame being received is a BSS beacon: if true, two actions are executed: i) one to activate a “switch back” countdown; and ii) `CHANGE_CHANNEL` to set up the Direct Channel.

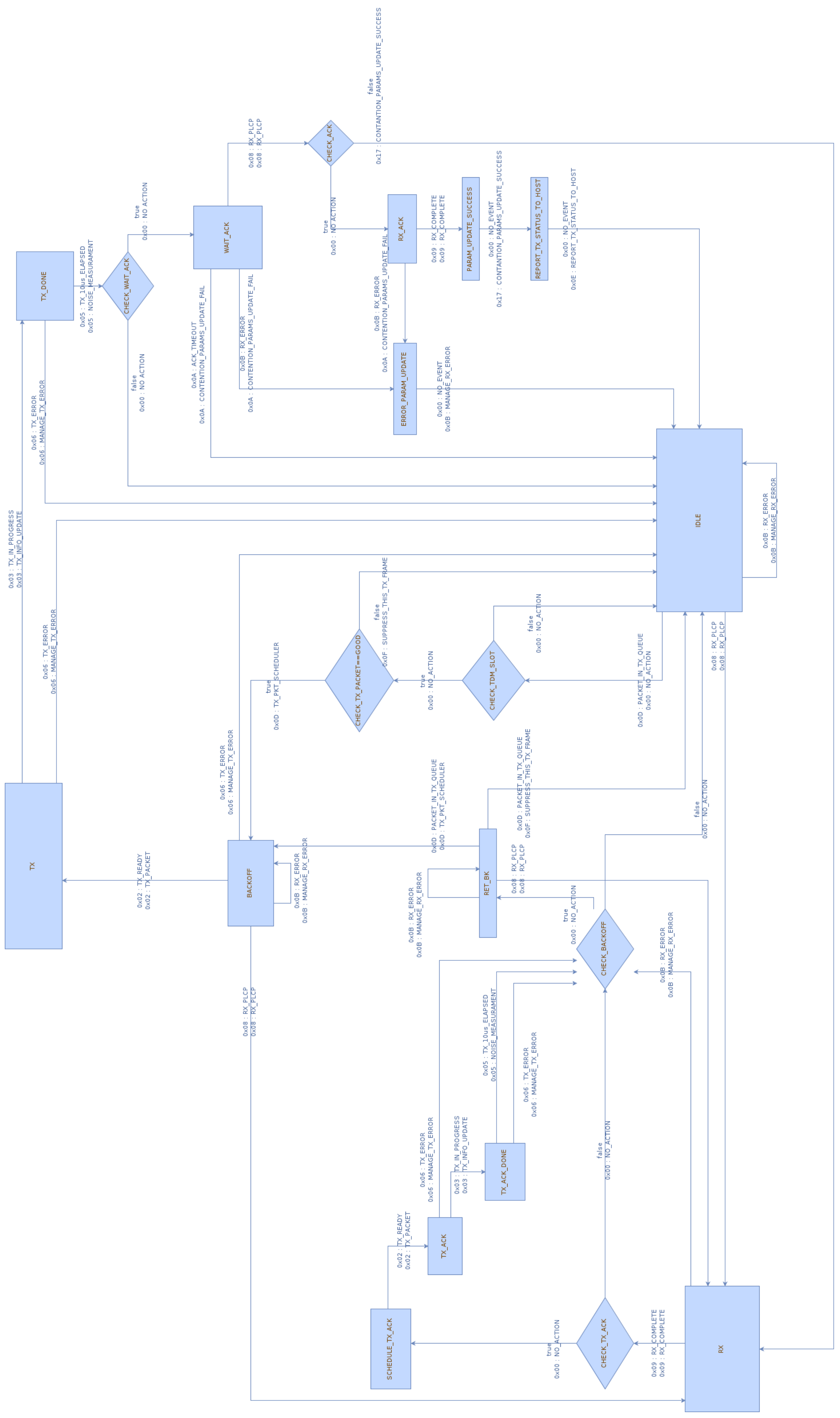
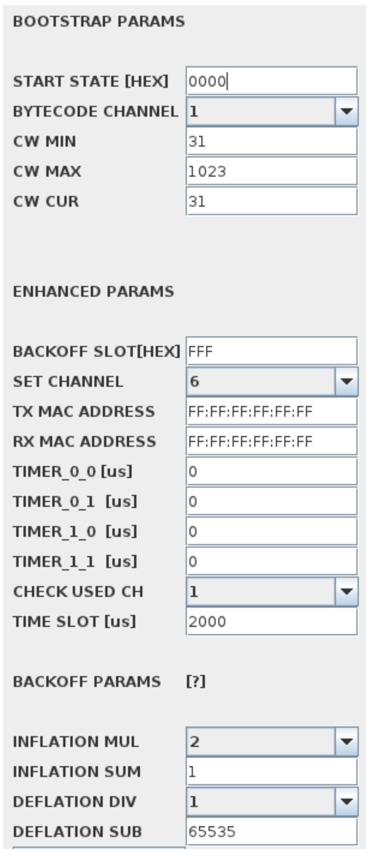


Figure 5.3: TDMA



BOOTSTRAP PARAMS

| | |
|---|---|
| START STATE [HEX] | 0000 |
| BYTECODE CHANNEL | 1 |
| CW MIN | 31 |
| CW MAX | 1023 |
| CW CUR | 31 |

ENHANCED PARAMS

| | |
|---|---|
| BACKOFF SLOT[HEX] | FFF |
| SET CHANNEL | 6 |
| TX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| RX MAC ADDRESS | FF:FF:FF:FF:FF:FF |
| TIMER_0_0 [us] | 0 |
| TIMER_0_1 [us] | 0 |
| TIMER_1_0 [us] | 0 |
| TIMER_1_1 [us] | 0 |
| CHECK USED CH | 1 |
| TIME SLOT [us] | 2000 |

BACKOFF PARAMS [?]

| | |
|---|---|
| INFLATION MUL | 2 |
| INFLATION SUM | 1 |
| DEFLATION DIV | 1 |
| DEFLATION SUB | 65535 |

Figure 5.4: TDMA State Parameters

### 5.4.3 State parameter in D(C)LS

Parameters used in D(C)LS are (see also Figures 5.7(b) and 5.7(b)):

- `PARAM_TX_DST_ADDR_0`: it is used by condition `TX_DST_ADDR == PARAM_TX_DST_ADDR_0` ;
- `PARAM_RX_SRC_ADDR_0`: it is used by condition `RX_SRC_ADDR == PARAM_RX_SRC_ADDR_0` ;
- `PARAM_SET_TIMER_0`: takes value `TIMER_0_0` as the Direct Link period expressed in microseconds;
- `PARAM_SET_CHANNEL`: used by action `CHANGE_CHANNEL` to define D(C)LS channel.

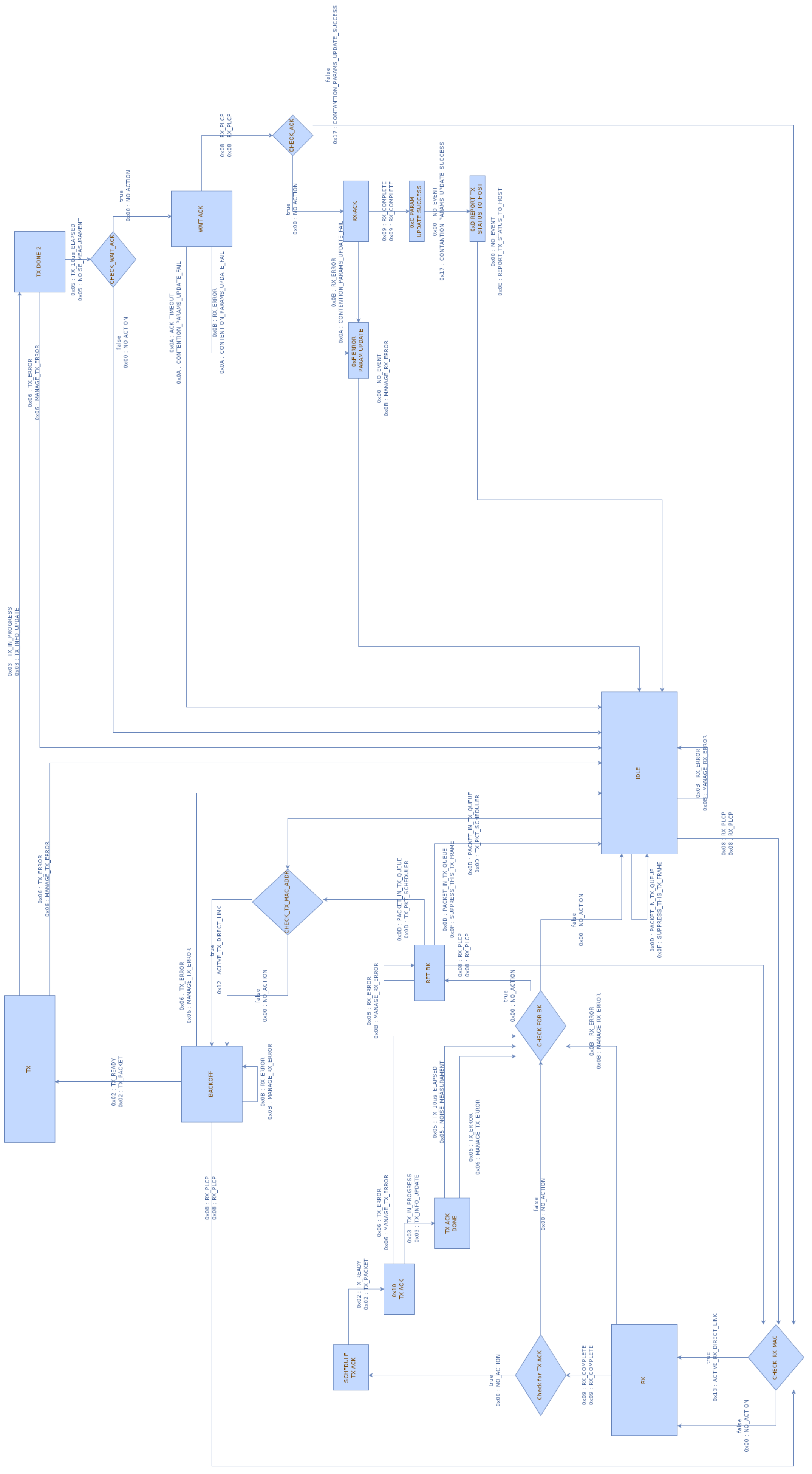


Figure 5.5: Direct Link Setup (DLS)

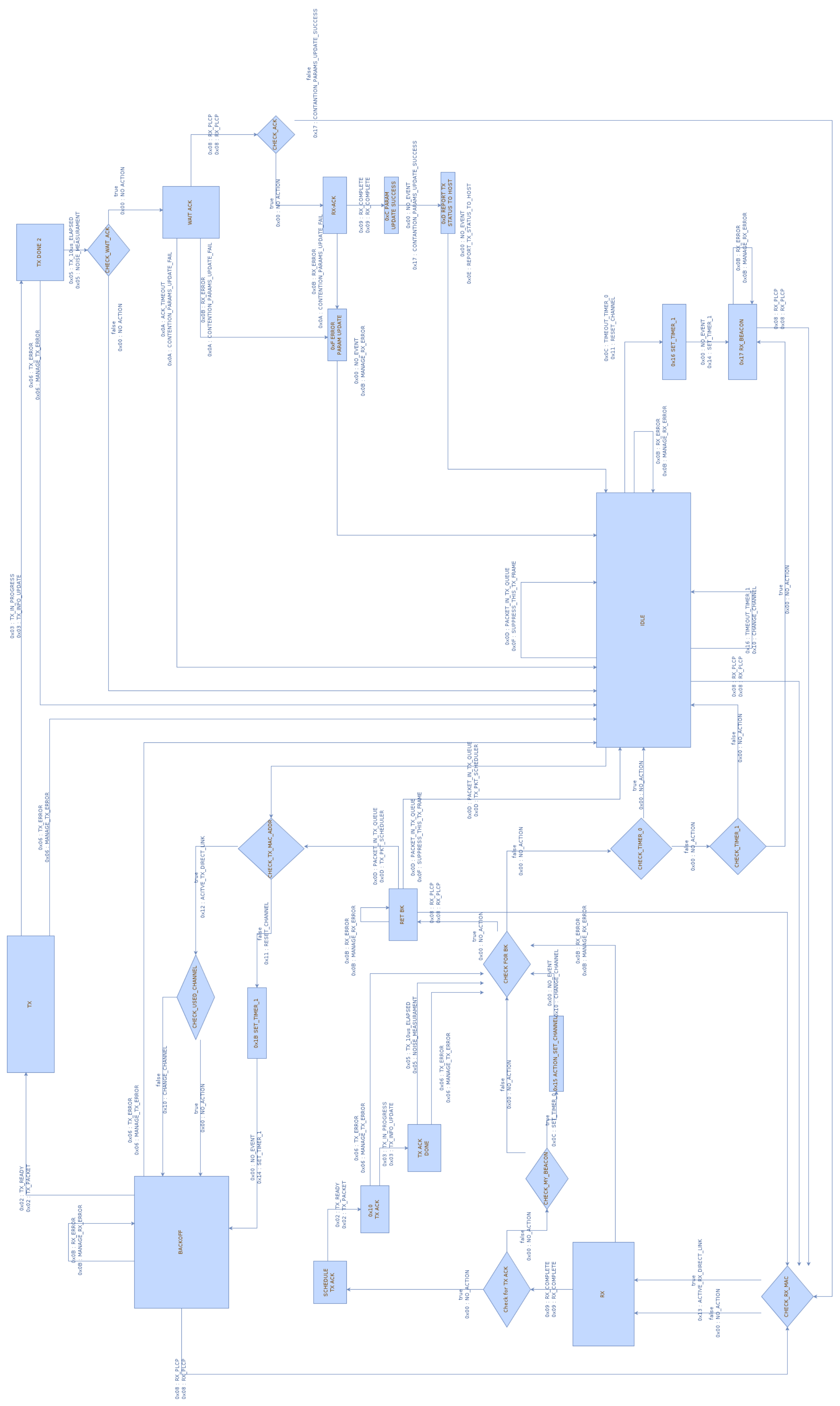
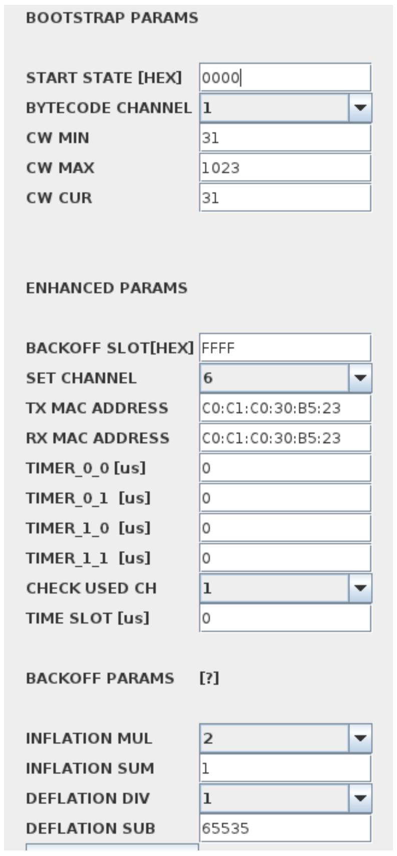
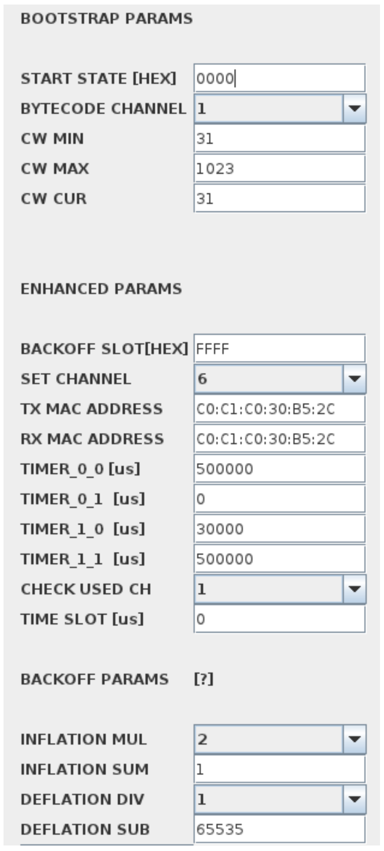


Figure 5.6: Direct Channel Link Setup (DCLS)



| | |
|---|---|
| BOOTSTRAP PARAMS | |
| START STATE [HEX] | 0000 |
| BYTECODE CHANNEL | 1 |
| CW MIN | 31 |
| CW MAX | 1023 |
| CW CUR | 31 |
| ENHANCED PARAMS | |
| BACKOFF SLOT[HEX] | FFFF |
| SET CHANNEL | 6 |
| TX MAC ADDRESS | C0:C1:C0:30:B5:23 |
| RX MAC ADDRESS | C0:C1:C0:30:B5:23 |
| TIMER_0_0 [us] | 0 |
| TIMER_0_1 [us] | 0 |
| TIMER_1_0 [us] | 0 |
| TIMER_1_1 [us] | 0 |
| CHECK USED CH | 1 |
| TIME SLOT [us] | 0 |
| BACKOFF PARAMS | [?] |
| INFLATION MUL | 2 |
| INFLATION SUM | 1 |
| DEFLATION DIV | 1 |
| DEFLATION SUB | 65535 |

(a)



| | |
|---|---|
| BOOTSTRAP PARAMS | |
| START STATE [HEX] | 0000 |
| BYTECODE CHANNEL | 1 |
| CW MIN | 31 |
| CW MAX | 1023 |
| CW CUR | 31 |
| ENHANCED PARAMS | |
| BACKOFF SLOT[HEX] | FFFF |
| SET CHANNEL | 6 |
| TX MAC ADDRESS | C0:C1:C0:30:B5:2C |
| RX MAC ADDRESS | C0:C1:C0:30:B5:2C |
| TIMER_0_0 [us] | 500000 |
| TIMER_0_1 [us] | 0 |
| TIMER_1_0 [us] | 30000 |
| TIMER_1_1 [us] | 500000 |
| CHECK USED CH | 1 |
| TIME SLOT [us] | 0 |
| BACKOFF PARAMS | [?] |
| INFLATION MUL | 2 |
| INFLATION SUM | 1 |
| DEFLATION DIV | 1 |
| DEFLATION SUB | 65535 |

(b)

Figure 5.7: DLS parameters 5.7(a), DCLS parameters 5.7(b)

# Chapter 6

# Byte-Code details

## 6.1 Introduction

Programmers should design MACs and customize their behavior using the graphical WMP-Editor: this tool really easy to use, in fact, enables rapid prototyping of Byte-Codes with few clicks of the user. Nevertheless, Byte-Codes can also be generated at hand, so in the present chapter we explain the Byte-Code structure, how to write or modify it at hand and finally we focus on the DCF Byte-Code that was introduced in Chapter 5, underlining the most meaningful details. Please take into consideration that writing the Byte-Code without the WMP-Editor is a difficult and error-prone task because programmers need a deep knowledge of the MAC-Engine and the way Byte-Code is interpreted and they must strictly adhere to the Byte-Code structure: for this reason this chapter should not be considered as a full guide to Byte-Code hand-writing but as an additional source of knowledge to better understand how the MAC-Engine works.

## 6.2 Structure of the Byte-Code

A Byte-Code defines a FSM, for this reason it contains a list of states and outgoing transitions, including a set of events, actions and conditions associated to each transition. The Byte-Code is hence a "description" of the FSM and it should be as compact as possible in order to meet the available memory limit. It must contain only ASCII characters including numbers and letters ranging from "A" to "F" so that strings can be translated into equivalent hexadecimal byte sequences. Comments can be inserted after character "#" and may contain all ASCII characters. Tag `000001` is empty and must be the first in the byte code, used as start delimiter; tag `000099` is the stop delimiter. The first part of byte code carrying information about the state machine is introduced by tag `000004` and contains the *state machine parameters*. The second part contains both *states* and *transitions*: each state is preceded by tag `000010` and the corresponding transitions by `000006`. Each transition block is terminated with special char `$`. In details:

- **000001** Delimits the beginning of the Byte-Code: what follows has to be injected into the WMP.
- **000004** It is a state parameter delimiter: informs the injector that next line is a state parameter. State parameters are sequentially written in a dedicated memory area. The injector starts from the position 0 and increment the position index for each state parameter. It is extremely important to have the state parameters in the same order the MAC-Engine expects them.
- **000006** It is a state transition delimiter: informs the injector that next line is a list of state transitions. Transitions are written in a sequential order. The injector starts from the position 0 at the beginning of the state transition area and increment position every time a new state transition is added. Byte-Code programmer has to keep track about the position of first outgoing transition for each state. It allows the correct editing of each state.
- **000010** State delimiter: it informs the injector that the next line has to be interpreted as a state. Such information is sequentially written in the memory area dedicated to store states. The injector keeps track of the starting position of each state and increments it at any state addition. The order of states is important for a correct workflow.

- **000099** Delimits the end of the Byte-Code: what follows has not to be injected into the WMP.

Data is aggregated in groups of 4 hex digits each. Since any hex digit has 4 bits, any group is 16 bit long. Values are represented in little endian, so the most significant byte is on the right of the least significant one: e.g., the following line

```
0E01010805082601010B010B3A01010D0200$
```

is split into groups of 4 hexadecimal digits:

```
0E01 - 0108 - 0508 - 2601 - 010B - 010B - 3A01 - 010D - 0200
```

inverted according to endianness:

```
010E - 0801 - 0805 - 0126 - 0B01 - 0B01 - 013A - 0D01 - 0002
```

and finally regrouped so that the output can be injected in the WMP and can be interpreted and executed by the MAC-Engine:

```
010E0801080501260B010B01013A0D010002
```

## 6.3 Byte-Code of the DCF State Machine

Table 6.1 contains and excerpt from the state machine of the DCF example. We dig into details in the following paragraphs.

**Byte-Code Header** First TAG is `000001` and indicates the beginning of the Byte-Code to inject.

**Byte-Code state parameters** The first "meaningful" section contains all the state machine parameters. Next TAG is `000004` and introduces the first state parameter that will be written in the first position of the dedicated memory area:

```
000004 #parameter @0x00
0100
```

where `000004` is the state parameter TAG, followed by a comment introduced by character "#". Then we have the actual value of the first parameter, in this case `PARAM_STATE_MACHINE_START`, that translates into `0001` according to endianness. Please note that the order of parameters is important and has to be taken into consideration by the part of the Byte-Code defining states and transitions to correctly adderess the corresponding parameters, e.g., `PARAM_STATE_MACHINE_START` should be addressed as first state parameter at address zero. For the sake of clarity the second parameter in the example

```
000004 #parameter @0x00
FFFF
```

will be addressed as the second parameter at address 1. We remark also that state parameters are 16 bits long. Following lines contain other state parameters but we skip the actual description.

**Byte-Code states and transitions** The description of states and transitions follows that of the state parameters: here each state is followed by its outgoing transitions. To better understand this section we focus on the definition of the IDLE state which is composed of the following fields:

- `000010`: state tag start
- `00F4`: translates into `F400` because of the used endianness that may be further decomposed as binary into

  ```
  1111 010 000000000
  ```

  where `1111` is a reserved sequence; `010` sets the number of outgoing transitions (three in this case, one is mandatory so `000` sets one transition) `000000000` is the position of the Transition area (zero in this case). It is computed in number of words (16bit per word) from the beginning of the memory region dedicated to transitions. The value can be obtained by considering the number of transitions written till the current one.

Table 6.1: Byte code representation of the “State Machine” explained.

| | |
|---|---|
| **#Header** | |
| `000001` | |
| **#state machine parameters** | |
| `000004` | |
| `0000` | |
| `000004` | |
| `FFFF` | |
| `000004` | |
| `0600` | |
| `000004` | |
| `FFFF` | |
| ... | |
| **# List of (state, transitions) items** | |
| `000010` | state tag |
| `00F4` | state IDLE #0x00 |
| `000006` | transition tag |
| `0000FF0802080000FF0B000B0000FF0D0B00$` | transition data |
| `000010` | state tag |
| `09F6` | state BACKOFF #0x01 |
| `000006` | transition tag |
| `0000FF0206020000FF0802080000FF0B010B0000FF060006$` | transition data |
| `000010` | state tag |
| `15F2` | state RX #0x02 |
| `000006` | transition tag |
| `0000FF0B030B0000FF090C00$` | transition data |
| `000010` | state tag |
| `1BF2` | state CHECK BACKOFF #0x03 |
| `000006` | transition tag |
| `0000FF1104000000FF000000$` | transition data |
| `00010` | state tag |
| `21F4` | state RET TO BACKOFF #0x04 |
| `000006` | transition tag |
| `0000FF0B040B0000FF0802080000FF0D0D00$` | transition data |
| ... | |
| `000010` | state tag |
| `54F2` | Virtual State #0x0B |
| `000006` | transition tag |
| `0000FF0E010D0000FF00000F$` | transition data |
| **# Terminator** | |
| `000099` | |

- `000006`: transition tag start
- `0000FF0802080000FF0B000B0000FF0D0B00$` this field represents three *Outgoing state transitions*, where each transition has a fixed length of 48bit (three 16bit words) and can in turn be further decomposed as follows (each subfield has already been converted according to endianness and splitted into components):

  1. Transition `0000 - 08 - FF - 08 - 02`

     `08` | event: `RX_PLCP`
     `FF` | no event parameter, no condition parameter
     `08` | action: `RX_PLCP`
     `02` | next state: `RX`.

     This means that after executing action `RX_PLCP`, state machine evolves to state `RX`.

  2. Transition `0000 - 0B - FF - 0B - 00`

     `0B` | event: `RX_ERROR`
     `FF` | no event parameter, no condition parameter

0B | action: `MANAGE_RX_ERROR`
00 | next state: `IDLE`.

In this case, when event `RX_ERROR` is raised, action `MANAGE_TX_ERROR` is executed and state machine goes back to state `IDLE`.

3. Transition `0000 - 0D - FF - 00 - 0B`

   0D | event: `PACKET_IN_TX_QUEUE`
   FF | no event parameter, no condition parameter
   00 | no action
   0B | next state: `#0x0B` - Virtual State.

   The last transition is triggered by event `PACKET_IN_TX_QUEUE`: it is worth noting that in this case no action is specified and that when the event is raised, state machine evolves to a *Virtual State*, in this case `#0x0B`. See the example below.

The following state is `BACKOFF`, according to value `F609` it is characterized by four transitions and, in fact, transition data `0000FF0206020000FF0802080000FF0B010B0000FF060006$` can be split after endianness translation as

```
0000 - 02 - FF - 02 - 06
0000 - 08 - FF - 08 - 02
0000 - B0 - FF - 0B - 01
0000 - 06 - FF - 06 - 00
```

**Byte-Code virtual states** Virtual states are not directly displayed by the GUI: they are, in fact, generated during the compilation process.

- `000010`: state tag start
- `54F2`: translates into `F254` because of the endianness which can be further decomposed into

  1111 001 001010100

  where again `1111` is a reserved sequence, `001` means two transitionts, and `001010100` sets the first transition at address `0x54` of the transition memory area.
- `0000FF0E010D0000FF00000F$` represents the two outgoing state transitions, namely

  1. Transition `0000 - 0E - FF - 0D - 01`

     0E | condition: `TX_PACKET == GOOD`
     FF | no condition parameter, no action parameter
     0D | action: `TX_PKT_SCHEDULER`
     01 | next state: `BACKOFF`.

     This transition is chosen if the condition `TX_PACKET == GOOD` is true: to verify the condition the MAC-Engine runs procedure `0x0E`, if returns true then action `TX_PKT_SCHEDULER` is executed and state machine evolves to state `BACKOFF`. If instead it returns false, then the MAC-Engine checks the next condition (here below).
  2. Transition `0000 - 00 - FF - 0F - 00`

     00 | condition: 00, ALWAYS TRUE
     FF | no condition parameter, no action parameter
     0B | action: `SUPPRESS_THIS_TX_FRAME`
     00 | next state: `IDLE`.

     Condition index `00` is a "non condition" meaning that it is always verified. In this transition the condition is used to always execute action `SUPPRESS_THIS_TX_FRAME` and evolve state machine to state `IDLE`.

**Byte-Code terminator** Last tag `000099` concludes the Byte-Code.

## 6.4 Binary-Byte-Code: details

The Binary-Byte-Codes of the two FSMs are stored in the shared memory of the reference NIC in range [0x0C20-0x0FFF], each one has a size of 496 bytes. For a deeper analysis we will refer to the second Binary-Byte-Code because it displays more easily. For the first Binary-Byte-Code worths the same notions. We recall here that the reference architecture is little-endian: memory dump will show odd and even address bytes swapped for the ease of comprehension. In the following we will analyze the Binary-Byte-Code of the DCF MAC. Each Binary-Byte-Code is organized in three distinct memory regions, which we countoured in red in Figure 6.1, they are:

- States region
- Transitions region
- State parameters region

### 6.4.1 States region

This region extends in range [0x0F90-0x0FFF], it counts 112 bytes. Each state is a 16bit value: up to 56 different states can be store, they are increasingly numbered beginning with 0. Each state encode the following fields:

- mask 0xF000: padding;
- mask 0x0E00: number of transitions from this state decreased by 1, for values between 0 (means 1 transition) and 6 (means 7 transitions). Value 7 means that the number of transitions is encoded in the Transitions region, at the end of the first seven transitions (details follow).
- mask 0x01FF: offset to the transition map from this state, referred to the Transitions region. This map extends for a number of transitions encoded in the previous field, or up to a marker if the previous field is set to 7. Nine bits would allow up to 512 transitions but the memory does not permit such plenty of transitions.

Considering Figure 6.1 the value associated to state 2, after endianness correction, becomes:
 0xF60F b(1111) (011) (000001111)
 and encode a state with tree condition, the first at offset 4 of the Transitions region.

### 6.4.2 Transitions region

Memory dedicated to transitions from the states extends over 816 bytes in range [0x0C60, 0x0F8E]. Each transition is encoded with 48 bit and holds the event or the condition, the action to be executed and the arrival state. Memory extension allow to represent up to 136 transitions (816 bytes / 6byte/cond). All transitions from a given state are contiguous and starts at address specified in the state definition; each state finally might have a different number of transitions. A transition which considers both event and condition, the latter towards a couple of arrival state, is managed with three different transition. The first is triggered by the event and leads to an intermediate state. The second and the third are function of the condition and they both lead to the final state.

Each transition encodes the following fields:

- mask FF FF 00 00 00 00: address of the procedure that checks for events and conditions. In the first case, code loops in the initial state until one event is verified, then execute the transition. In the second case, instead, code immediately evaluates the condition and jumps to one of the two final states. It is worth noting that the code involved in these checks is dynamically generated by Byte-Code-Manager according to the second field (below): this improves the execution speed of code involved in events and conditions checking. Note also that in Figure 2.4 this field is empty because code will be generated only at injection time.
- mask 00 00 F0 00 00 00: condition/event parameter, see the following one.
- mask 00 00 0F 00 00 00: action parameter. This field and the previous one allows to pass some data chosen by users to the procedure that implements the condition/event checking. These parameters increase the flexibility of events, actions and conditions
- mask 00 00 00 FF 00 00: index of the condition/event. It is filled during the definition of the FSM with the unique id of the condition/event. It is used by the injection code to get the address of the procedure to set in the field mask FF FF 00 00 00 00, in this way the MAC-Engine has the address of event and condition directly, this improves the execution speed of code involved in events and conditions checking.

- mask 00 00 00 00 FF 00: index of the arrival state. Used by the WMP to select the final state after having checked the condition/event and executed the action.
- mask 00 00 00 00 00 FF: action index. This is filled during the FSM definition with the id of the action and used by the WMP to get the address of the procedure to execute after the event or condition associated to the transition has been verified.

The tree transitions from state 2, considered in the previous example are stored starting at address 15, are underlined in Figure 6.1 and they extend in $6 * 4 = 24$ bytes.

- transition 0: CB 00 FF 02 04 02
- transition 1: 26 01 FF 0B 02 0B
- transition 2: 0E 01 FF 08 03 08
- transition 3: EB 00 FF 06 00 06

First transition has the following meaning

1. CB 00, event address
2. F, parameter for condition/event, F means no parameter
3. F, parameter for action, F means no parameter
4. 02, identify event `TX_READY`
5. 04, arrival state for the transition if event verify
6. 02, identify action `TX_PACKET`

### 6.4.3 State parameters region

This region extends on 64 bytes in the range [0x0C60-0x0C20] and contains two kind of parameters, those associated to the entire Byte-Code and those used in events, conditions and actions.

Those in the first set are used during the Byte-Code bootstrap and after a Byte-Code switch, in the reconfiguration phase. All parameters can be set using an interface of the WMP-Editor.

```
0x0C20:  0000 FFFF 0100 FFFF FFFF FFFF FFFF FFFF
0x0C30:  FFFF 0000 0000 0000 0000 0000 0000 0000
0x0C40:  0000 0000 0000 0000 0000 0100 0000 0100
0x0C50:  0000 0000 0000 0000 0000 0000 0000 0000
```

```
0x0C60:  2601 FF0B 000B 3A01 FF0D 0100 0E01 FF08
0x0C70:  0308 5B01 010E 020D 0000 FF00 000F CB00
0x0C80:  FF02 0402 2601 FF0B 020B 0E01 FF08 0308
0x0C90:  EB00 FF06 0006 1801 FF09 0709 2601 FF0B
0x0CA0:  0B0B EB00 FF06 0006 D100 FF03 0A03 7001
0x0CB0:  FF10 0C00 0000 FF00 0000 1801 FF09 0009
0x0CC0:  2601 FF0B 000B 6901 FF0F 0800 0000 FF00
0x0CD0:  0B00 CB00 FF02 0802 EB00 FF06 0B06 D100
0x0CE0:  FF03 0903 DD00 FF05 0B05 EB00 FF06 0B06
0x0CF0:  EB00 FF06 0006 DD00 FF05 0505 7701 FF11
0x0D00:  0E00 0000 FF00 0000 0E01 FF08 0D08 1F01
0x0D10:  FF0A 000A 2601 FF0B 000B CF01 FF19 0600
0x0D20:  0000 FF00 0300 0E01 FF08 0308 2601 FF0B
0x0D30:  0E0B 3A01 FF0D 0100 0100 1300 0000 0100
0x0D40:  0D10 7E01 0112 1512 0000 0100 1111 A001
0x0D50:  0113 0513 0000 0100 0500 0000 0000 0314
0x0D60:  0E01 0108 1008 2601 010B 120B FFFF B301
0x0D70:  0115 0100 BE01 0117 1200 0000 0100 0100
0x0D80:  0000 0100 1214 C301 0118 0310 0000 0100
0x0D90:  0300 0E01 0108 1708 1F01 010A 010A 2601
0x0DA0:  010B 010B CF01 0119 0800 0000 0100 1000
0x0DB0:  3A01 010D 0200 0E01 0108 1008 2601 010B
0x0DC0:  180B D501 011A 0200 0000 0100 0100 0000
0x0DD0:  0000 0000 0000 0000 0000 0000 0000 0000
0x0DE0:  0000 0000 0000 0000 0000 0000 0000 0000

                      .     .     .     .
                      .     .     .     .
                      .     .     .     .

0x0F80:  0000 0000 0000 0000 0000 0000 0000 0000
```

Transition of the state 2

```
         State 0  State 1 State 2  State 3   ......................
0x0F90:  00F4 09F2 0FF6 1BF2 21F2 27F2 2DF2 33F2
0x0FA0:  39F4 42F2 48F2 4EF2 54F4 5DF2 63F4 71F2
0x0FB0:  77F2 7DF0 80FE 87F4 90F0 93F2 99F4 A2F2
0x0FC0:  A8F4 B1F2 0000 0000 0000 0000 0000 0000
0x0FD0:  0000 0000 0000 0000 0000 0000 0000 0000
0x0FE0:  0000 0000 0000 0000 0000 0000 0000 0000
0x0FF0:  0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 6.1: byte-code region

# Chapter 7

# Byte-Code-Manager

Byte-Code-Manager is a software tool that can be used to inject a Byte-Code state machine into the WMP and in general to interact with the WMP system. Though users may run it directly from the command line, in server mode the tool waits for commands from the network.

## 7.1 bytecode-manager options list

```
root@sta01# bytecode-manager
-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
WMP bytecode-manager byte-code injection
Usage: bytecode-manager [OPTIONS]
   -h                         Print this help text
   -l <#>                     LOAD Bytecode in specified # value (1 or 2)
   -m <name-file>             LOAD Bytecode state-machine bytecode file
   -u                         used with the options -a -l and -m force
                              the load of bytecode
   -a <#>                     Activate specified bytecode (1 or 2)
   -t <time>                  Timed Bytecode Activation [value in sec]
   -d <delay>                 Delayed Bytecode Activation in microsecond
   -f <time>                  Return the absolut time for precise
                              equal activation [value in sec]
   -r                         reset activate and deactive condition Bytecode
   -v                         view active and deactive condition Bytecode
   -c <ip address>            IP address to server station Start in client mode
   -g <name-file>             bytecode to send
   -s <interface to listen>   SERVER MODE
   -p <port number>           In server mode or client mode select specific port,
                              if not use default port is 9898
   -e <on><off>               active or deactive state debug
   -x <1,2,3>                 Show Registers (1), Share Memory(2) or both(3)
```

## 7.2 Stand-alone Operation Mode

This section describes the following stand-alone functions:

- Byte-Code injection and activation
- timed activation, delayed activation

### 7.2.1 Byte-Code injection and activation

The main feature of the bytecode-manager is Byte-Code injection. A Byte-Code can be injected into one of two different Byte-Code areas: though area 1 is filled at startup with a default Byte-Code, it can be replaced with a new one.

This command injects the Byte-Code stored in file `mycode.txt` into area 2:

```
root@sta01# bytecode-manager -l 2 -m mycode.txt
```

and this one activates it:

```
root@sta01# bytecode-manager -a 2
```

To activate back the Byte-Code in area 1:

```
root@sta01# bytecode-manager -a 1
```

### 7.2.2 Delayed Byte-Code switching

Byte-Code switching can be scheduled at a given time in the future, by either defining a delay or an absolute time: in both cases the event is handled by the WMP by periodically checking the internal clock. Since all stations in a given BSS synchronize their internal clock with that of the Access Point, the second mechanism allows to switch the Byte-Code on several station at the same time.

This command schedules a Byte-Code switch after twenty seconds:

```
root@sta01# bytecode-manager -t 20
```

This command schedules a Byte-Code switch at a given time:

```
root@sta01# bytecode-manager -d <value-time-us>
```

where <value-time-us> is an accurate clock reference expressed in microsecond. When the internal clock reaches <value-time-us>, the WMP deactivates the current active Byte-Code and activate the other one.

Again bytecode-manager can be used to get the <value-time-us> corresponding to a given delay:

```
root@sta01# bytecode-manager -f <delay-in-second>
```

The output value is expressed in microseconds and is computed by summing the input <delay-in-second> to the internal clock. For example, if we want to switch the Byte-Code on all stations in 12 seconds we should first get the reference time on one station, i.e.,

```
root@sta01# bytecode-manager -f 12

-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
Selected find absolute time
Current work mode : "local"
------------------------------------
Calculation value of activation delay
time stamp : 3076057456
------------------------------------
```

Then we must run option `-d` on all stations using the `time stamp` value that was returned (`3076057456`).

To cancel timers, run:

```
root@sta01# bytecode-manager -r
```

To display information about timers run:

```
root@sta01# bytecode-manager -v
-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
Current work mode : "local"
Selected view

-------------------------------------
CURRENT BYTECODE = 1
Control Value    = 0x0000
Timer Not Active
Delay Not Active
-------------------------------------
```

### 7.2.3 Client-Server Operation Mode

This section describes how to setup a Client-Server WMP configuration service using the Byte-Code-Manager tool. First, the server should be started on a WMP station (e.g., `sta01`):

```
root@sta01# bytecode-manager -s

-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
Current work mode : "server"
Starting bytecode Manager in SERVER mode, listen on port 9898
```

By default TCP port 9898 is used but can be optionally changed with option `-p`. Once the server is running on `sta01`, all commands described in section 7.2 can be run from another machine (e.g. `sta02`:

```
root@sta02# bytecode-manager -c sta01 -v

-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
Current work mode : "client"
recvBuffer = |v=OK
-------------------------------------
CURRENT BYTECODE = 1
Control Value    = 0x0000
Timer Not Active
Delay Not Active
-------------------------------------
```

Also Byte-Code images can be transferred using the client-server model: to send a Byte-Code file table to a remote machine use this command:

```
root@sta02# bytecode-manager -c sta01 -g /path/to/mybytecode.txt
-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
file /path/to/mybytecode.txt found
Current work mode : "client"
filename sent=dcf_fix_bk.txt
recvBuffer = OK-FIN
```

Once the Byte-Code is transferred it can be remotely loaded and activated: e.g.,

```
root@sta02# bytecode-manager -c sta01 -l 2 -m mybytecode.txt
-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
Current work mode : "client"
recvBuffer = |l=OK|m=OK
```

The MAC-Engine implements blocking techniques during Byte-Code loading but, in some case, is necessary to force Byte-Code injection without any safety procedure. Byte-Code-Manager provides an option that allows hard injection of the Byte-Code, useful when a Byte-Code doesn't work correctly and, for example, loops in dead events. The option is `-u`

```
root@sta02# bytecode-manager -c sta01 -l 2 -m mybytecode.txt -u
```

Finally, there are a few options to debug the WMP, by getting a dump of the MAC-Engine registers:

```
root@sta02# bytecode-manager -x 1
-------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
-------------------------------------------
```

```
Current work mode : "local"
Link registers:
lr0: 0AD1  lr1: 0B11  lr2: 0149  lr3: 0261
Offset registers:
off0: 0180  off1: 0204  off2: 039F  off3: 039F
off4: 0130  off5: 045F  off6: 3010
General purpose registers:
r00: 0001  r01: 0000  r02: 0001  r03: 000F
r04: 0000  r05: 0020  r06: 0007  r07: 0004
r08: 001F  r09: 0000  r10: 0001  r11: 053E
[...cut...]
r52: 0001  r53: 000F  r54: 01FF  r55: 0000
r56: 0441  r57: 0418  r58: 0000  r59: 0708
r60: 0000  r61: 0000  r62: 0000  r63: 0000
```

and of the MAC-Engine memory:

```
root@sta02# bytecode-manager -x 2
bytecode-manager -x 2
--------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
--------------------------------------------
Current work mode : "local"
SHM dump 2

Shared memory:

0x0000: 9A01 7008 FFFF 0A7C 0000 0000 C000 0A00
0x0010: 1400 0000 8000 0900 4700 4700 8301 6400
0x0020: 3009 C0FC 0000 0000 0000 0000 0000 0000
[...cut...]
```

To get a snapshot of the evolution of the state trace, first activate it:

```
root@sta02# bytecode-manager -e on
--------------------------------------------
WMP Bytecode Manager V 0.1 - 2012
--------------------------------------------
Current work mode : "local"
Selected state-debug
```

Then dump the MAC-Engine memory and check memory in range `[0x0E00 -0x0F60]`.

```
0x0E00: 0000 0000 0000 0000 0000 0000 0000 0000
0x0E10: 0000 FF00 0D00 FF00 FF08 0500 FF09 0900
0x0E20: FF00 0D00 FF00 0100 FF08 0500 FF09 0900
[...cut...]
0x0F30: 0100 FF08 0500 FF09 0900 FF00 0D00 FF00
0x0F40: 0100 FF08 0500 FF09 0900 FF00 0D00 FF00
0x0F50: 0100 FFFF 0000 0000 0000 0000 0000 0000
```

Chapter 8

# Usage scenarios and examples

This chapter demonstrates how to setup and manage Byte-Codes on WMP enabled stations. This is not a guide for writing a new Byte-Code: existing Byte-Codes that we published on the web site will be used. Expert users could modify the released Byte-Codes by using the WMP-Editor tool as explained in the previous Chapters.

## 8.1 Basic usage at a glance

Pre-requisite for the following examples is the availability of a compatible Broadcom card: a 4311v1 or 4318 chipset should be used, if you face issues with your NIC, it probably means that its chipset is not yet supported. In any case do not hesitate contacting us. You can find instructions for setting up the driver and the original firmware required to use module `b43` on these NICs here:

- `http://wireless.kernel.org/en/users/Drivers/b43#firmwareinstallation`

We will now replace the original firmwares with the WMP software:

- Get the WMP tools by downloading from here:
  - `http://wmp.tti.unipa.it/index.php/downloads`
- The WMP tools include:
  - WMP main firmware
  - Byte-Code-Manager
  - WMP-Editor
  - example Byte-Codes (DCF, TDM, DLS)
- The WMP main firmware is composed by three file:
  - `ucode5.fw` binary firmware;
  - `b0g0bsinitvals5.fw` initial values;
  - `b0g0initvals5.fw` initial values.
- copy the three files in folder `/lib/firmware/b43` on your station, reload the `b43` kernel module and reconfigure the NIC:
  - `$: cp ucode5.fw b0g0bsinitvals5.fw b0g0initvals5.fw /lib/firmware/b43`
  - `$: rmmod b43`
  - `$: modprobe b43 qos=0`
  - `$: ifconfig wlan0 192.168.1.2 netmask 255.255.255.0 up`
- connect to an Access Point, e.g.:
  - `$: iwconfig wlan0 essid YOUR-ESSID`

**NB: DO NOT FORGET** parameter `qos=0`, the binary firmware does not yet support QOS
**NB2: USE THE FIRMWARE** only for stations since it does not support AP mode.
**NB3: KERNEL MODULE `b43` SHOULD** have been compiled with options "debugfs" and "b43 debugging" enabled: read more information on how enabling these options in the Appendix.

- Byte-Code-Manager is the tool for managing the WMP: test everything is working as expected by activating the Byte-Code in position 2,
  - `$: ./bytecode-manager -a 2`
- Use WMP-Editor to modify or create a new Byte-Code from scratch, i.e., "`new-byte-code`", then use Byte-Code-Manager to load different Byte-Codes (read instructions for use the Byte-Code-Manager). In the following we will activate back Byte-Code in position 1, replace the Byte-Code in position 2 with the standard DCF code, activate it, replace and the firmware in position 1 with the one you modified or created from scratch and finally activate it:
  - `$: ./bytecode-manager -a 1`
  - `$: ./bytecode-manager -l 2 -m DCF`
  - `$: ./bytecode-manager -a 2`
  - `$: ./bytecode-manager -l 1 -m new-byte-code`
  - `$: ./bytecode-manager -a 1`

## 8.2 Advanced usage

To get the source code of the binary WMP firmware please send an email to the "WMP-team". Please report to the team any change for addition to the repository. Tools for handling firmware (b43-tools) can be found here

- `http://git.bues.ch/gitweb?p=b43-tools.git`

You have to compile the debugging software `b43-fwdump` and the assembler `b43-asm`. The former can be used to dump the memory of the wireless card, shared memory and registers, and analyze in real time how the WMP works. You can also see the MAClet hexadecimal representation in the NIC shared memory. The latter is required to modify the MAC-Engine source code, e.g., to add new events and conditions, and to compile it into a runnable binary firmware. Remember that if you add new features you have also to modify the WMP-Editor so that new features can be easily configured on the graphical editor. For more information on the WMP read the specific sections.

## 8.3 Quick usage examples

Below you find a series of examples of use of WMP. For each scenario, please follow Section 8.1 down to the point when you connected the NIC to the Access Point. In the following we assume AP address is `192.168.1.1`.

### 8.3.1 Example 1: Byte-Code load and activation

- Ping the AP:
  - `$: ping 192.168.1.1`
- Display the current Byte-Code:
  - `$: ./bytecode-manager -v`
- Load Byte-Code named "maclet.txt" in a different position, e.g., if the result of the previous command was 1, then load this new firmware in position 2:
  - `$: ./bytecode-manager -l 2 -m maclet.txt`
- Activate this new Byte-Code:
  - `$: ./bytecode-manager -a 2`

### 8.3.2 Example 2: delayed Byte-Code activation

- Ping the AP:
  - `$: ping 192.168.1.1`
- Display the current Byte-Code:
  - `$: ./bytecode-manager -v`
- If the result of the last command was 1, then load new Byte-Code named "byte-code.txt" into position 2:
  - `$: ./bytecode-manager -l 2 -m byte-code.txt`
- Set the timer to switch the Byte-Code after 40 seconds:
  - `$: ./bytecode-manager -t 40`
- Check the active timer and the activated Byte-Code:
  - `$: ./bytecode-manager -v`
- Repeat the last command after 40 seconds to check that the Byte-Code has been actually changed and that the timer has been reset.

### 8.3.3 Example 3: accurate delayed Byte-Code activation

- Ping the AP:
  - `$: ping 192.168.1.1`
- Display the current Byte-Code:
  - `$: ./bytecode-manager -v`
- If the result of the last command was 1, then load new Byte-Code named "byte-code.txt" into position 2:
  - `$: ./bytecode-manager -l 2 -m byte-code.txt`
- Find the timestamp in the future (50 seconds) with the absolute reference provided by the card:
  - `$: ./bytecode-manager -f 50`
- The output should be something similar:

```
Selected find-delay
Select only find-delay
*************************************
Calculation value of activation delay
time stamp : 3292517609
*************************************
```

- Setup the switch time using the result of the last command:
  - `$: ./bytecode-manager -d 3292517609`
- Check the active timer and the activated Byte-Code:
  - `$: ./bytecode-manager -v`
- Repeat the last command after 50 seconds to check that the Byte-Code has been actually changed and that the timer has been reset.

### 8.3.4 Example 4: remote Byte-Code switching

We set up an environment in which there are two stations ( STA1 - STA2) connected to an Access Point (AP), both stations use the WMP firmware (no requirement on the AP wireless setup). We use the AP to control stations with option `-c|-client` of the Byte-Code-Manager.

On the AP we use the following configuration values:

- IP address: `192.168.1.50`
- netmask: `255.255.255.0`
- essid: `wmp-ap`

Set up STA1 as follows

- `$: modprobe b43 qos=0`
- `$: ifconfig wlan0 192.168.1.1`
- `$: netmask 255.255.255.0`
- `$: iwconfig wlan0 essid wmp-ap`
- `$: bytecode-manager -s wlan0`

Set up STA2

- `$: modprobe b43 qos=0`
- `$: ifconfig wlan0 192.168.1.2`
- `$: netmask 255.255.255.0`
- `$: iwconfig wlan0 essid wmp-ap`
- `$: bytecode-manager -s wlan0`

At the AP use Byte-Code-Manager to activate the firmware at position 1 in both stations:

- `$: ./bytecode-manager -c 192.168.1.1 -a 1`
- `$: ./bytecode-manager -c 192.168.1.2 -a 1`

Then send the new Byte-Code called “byte-code.txt” to both stations:

- `$: ./bytecode-manager -c 192.168.1.1 -g byte-code.txt`
- `$: ./bytecode-manager -c 192.168.1.2 -g byte-code.txt`

Load Byte-Code “byte-code.txt” in position 2 on both stations:

- `$: ./bytecode-manager -c 192.168.1.1 -l 2 -m byte-code.txt`
- `$: ./bytecode-manager -c 192.168.1.2 -l 2 -m byte-code.txt`

and finally activate the loaded Byte-Code in both stations:

- `$: ./bytecode-manager -c 192.168.1.1 -a 2`
- `$: ./bytecode-manager -c 192.168.1.2 -a 2`

### 8.3.5 Example 5: remote Byte-Code delayed switching

Follow 8.3.4 down to the point when you loaded Byte-Code named "byte-code.txt" in position 2 on both stations, without activating it, i.e., down to here

- `$: ./bytecode-manager -c 192.168.1.1 -l 2 -m byte-code.txt`
- `$: ./bytecode-manager -c 192.168.1.2 -l 2 -m byte-code.txt`
- Now find the timestamp of 90 second in the future with the absolute reference provided by STA2:
  - `$: ./bytecode-manager -c 192.168.1.2 -f 90`
- You should get something similar:

```
*************************************
Calculation value of activation delay
time stamp : 3292517609
*************************************
```

- Now setup the switch time on both station using the output produced by the last command:
  - `$: ./bytecode-manager -c 192.168.1.1 -d 3292517609`
  - `$: ./bytecode-manager -c 192.168.1.2 -d 3292517609`

All the stations in the network will switch to the new Byte-Code exactly at the same time.

# Appendix A

# APPENDIX

## A.1 Overview

There are no known compatibility issues between the WMP firmware and all versions of the Linux kernel and the b43 driver: users are free to use their preferred Linux Distribution. Nevertheless we want to share some tricks that can improve usability and stability of the testbed environment.

## A.2 Modify the kernel to improve usability and stability of the testbed environment

### A.2.1 Avoid deassociation

To prevent stations deassociating from the AP because of issues with reception of beacon frames and/or exchange of management frames, apply the following changes to the kernel source.

#### A.2.1.1 For older kernel versions

- Edit file `net/mac80211/mlme.c`, in function `ieee80211_associated()` find the following code:

```
if (ifsta->flags & IEEE80211_STA_PROBEREQ_POLL) {
        printk(KERN_DEBUG "%s: No ProbeResp from "
        "current AP %pM - assume out of "
        "range\n",  sdata->dev->name, ifsta->bssid);
        disassoc = 1;
```

- Comment assignment `//disassoc = 1;`, then recompile the kernel and reinstall the b43 module.

#### A.2.1.2 For more recent kernel versions

- Edit file `net/mac80211/mlme.c`, find the following snippet of code:

```
/*  * Time we wait for a probe response after sending
    * a probe request because of beacon loss or for
    * checking the connection still works. */
static int probe_wait_ms = 500;
module_param(probe_wait_ms, int, 0644);
MODULE_PARM_DESC(probe_wait_ms,
"Maximum time(ms) to wait for probe response"
" before disconnecting (reason 4).");
```

- Replace wait time from 500ms to 10000ms. This should be enough to avoid issues.
- To completely avoid deassociation, in the same source file find this snippet of code:

```
else {
/*  *We actually lost the connection ... or did we?
```

```
    *Let's make sure! */
wiphy_debug(local->hw.wiphy, "%s: No probe response from AP %pM"
            " after %dms, disconnecting.\n",
            sdata->name, bssid, probe_wait_ms);
ieee80211_sta_connection_lost(sdata, bssid);
```

- Comment line `ieee80211_sta_connection_lost(sdata, bssid);`.

### A.2.2 Solving the throughput problem of Iperf

Iperf is a simple program to measure the effective bandwidth between a pair of peers by using saturating traffic such as UDP. For this reasons Iperf is a great tool to test the Byte-Code developed in the WMP. Unfortunately, recent versions of Linux kernel had a bug that caused the DMA subsystem of the b43 driver to keep sending data to the NIC even when the output queue is full, leading to great losses and wrong throughput reports at sender. To avoid this problem make sure your kernel contains the following patch:

- `http://lists.infradead.org/pipermail/b43-dev/2011-December/002240.html`

## A.3 How to enable "debugfs" and "b43 debugging" in the Kernel

As already mentioned Byte-Code-Manager needs these options enabled in the Kernel: run `make menuconfig` and

- for "debugfs"

```
Kernel hacking --->
-*- Debug Filesystem
[ ] Run 'make headers_check' when building vmlinux
[ ] Enable full Section mismatch analysis
[*] Kernel debugging
```

- for "b43 debugging"

```
Device Drivers --->
-*- Network device support --->
[*] Wireless LAN --->
<M> Broadcom 43xx wireless support (mac80211 stack)
[ ] Broadcom 43xx PCMCIA device support
[ ] Broadcom 43xx SDIO device support (EXPERIMENTAL)
[*] Support for 802.11n (N-PHY) devices (EXPERIMENTAL)
[*] Support for low-power (LP-PHY) devices (EXPERIMENTAL)
[*] Broadcom 43xx debugging
[ ] Force usage of PIO instead of DMA
```

- Moreover, if missing, you need debugfs mounted in directory `/sys/kernel/debug`. To this end run:

```
$: mount -t debugfs none /sys/kernel/debug
```